RICE UNIVERSITY

Array Syntax Compilation
and Performance Tuning

by

Yuan Zhao

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Ken Kennedy, Chair
John and Ann Doerr University Professor
Computer Science

Jack Dongarra
University Distinguished Professor
Computer Science
University of Tennessee

John Mellor-Crummey
Associate Professor
Computer Science

Robert Fowler
Director of HPC Research
Renaissance Computing Institute

Danny C. Sorensen
Noah Harding Professor
Computational and Applied Mathematics

Houston, Texas
December, 2006
Array Syntax Compilation and Performance Tuning

Yuan Zhao

Abstract

Array syntax adds expressive power to a language by providing operations on and assignments to array sections. Thus it allows programmers to write clear and concise code. However, state-of-the-art vendor compilers fail to efficiently map array statements to underlying architectures for high performance. The inefficiency is caused by ineffectively solving the following three technical problems: (1) reducing the size of allocated temporary array; (2) extending solutions to the evolving architectures; (3) applying loop fusion to multiple array statements. Finding solutions to these problems is important because otherwise array syntax, though a high-level language feature, may not be widely used by application developers.

To address the above problems, this research first develops a novel strategy that minimizes the allocated temporary arrays using loop alignment and loop skewing on scalar processors, thereby reducing memory traffic and improving cache utilization. It then extends the minimization strategy to exploit the increasing on-chip parallelism on evolving architectures that offer vector (e.g., SSE and Altivec) and multi-core (e.g., CELL) capabilities. In addition, new techniques boost performance by improving data alignment and managing data movement, both of which are important on these new architectures. Last, this dissertation parameterizes loop fusion for performance tuning and explores the properties of the space of all possible loop fusion configurations, to expedite performance tuning of loop fusion for increasing data reuse across multiple array statements.
These transformations and optimizations are implemented in a source-to-source research compiler with extensions to target short vector processors and CELL processor. Experiments show that array statements compiled with our strategy run as much as two times faster than those compiled directly by vendor compilers. Our exploration of loop fusion parameter space identifies good candidates for heuristic searching and space pruning, which are essential to make the performance tuning process practical.

In summary, this dissertation demonstrates that advanced compilation techniques can significantly improve the performance of programs written in array syntax upon current state-of-the-art implementation across a variety of architectures, including the latest multi-core processors with vector capabilities.
Acknowledgments

I would like to thank my advisor, Dr. Ken Kennedy, for his constant help, guidance, and support in the development of this dissertation. Sincere thanks go to Dr. John Mellor-Crummey, Dr. Dan Sorensen, Dr. Jack Dongarra and Dr. Robert Fowler for being members of my thesis committee. Special thanks go to Dr. Qing Yi, Dr. Dan Quinlan, and Dr. Richard Vuduc for their help.

Finally, I thank my parents and my brother for their support.
Contents

1 Introduction ........................................... 1
   1.1 Array Syntax: Syntax, Semantics and Implementation ............ 1
   1.2 Thesis ............................................. 4
   1.3 Compiling Array Syntax for High Performance .................... 4
       1.3.1 Architectures .................................... 5
       1.3.2 Inter-Statement Data Reuse ........................... 10
       1.3.3 Summary ......................................... 11
   1.4 Structure of Dissertation ................................... 12

2 Compiling for Scalar Processors .................................. 13
   2.1 Introduction ......................................... 14
       2.1.1 Scalarization Problem on Scalar Processors .............. 14
       2.1.2 Scalarization Dependence Matrix ........................ 15
   2.2 Related Work .......................................... 17
   2.3 Loop Alignment ........................................ 20
       2.3.1 Loop Alignment in Scalarization ....................... 20
       2.3.2 Applicability and Profitability of Loop Alignment ......... 22
       2.3.3 Loop Alignment vs. Input Prefetching ................... 24
2.3.4 Lazy Loop Alignment .................................................. 25
2.4 Loop Skewing ................................................................. 28
  2.4.1 Loop Skewing in Scalarization ................................... 28
  2.4.2 Profitability and Applicability of Loop Skewing .......... 30
  2.4.3 Summary of Scalarization .......................................... 32
2.5 Application: Stencil Computation .................................... 32
  2.5.1 Minimal Temporary Storage Requirements for Two Dimensional
         Stencil Computations .................................................. 34
  2.5.2 Minimal Temporary Storage Requirements for Multi-Dimensional
         Stencil Computations .................................................. 43
  2.5.3 Optimal Scalarization for Stencil Calculations ............ 50
2.6 Experiments ..................................................................... 54
  2.6.1 Loop Alignment .......................................................... 55
  2.6.2 Commercial Compilers ............................................... 58
  2.6.3 Fine Tuning Loop Alignment ....................................... 60
  2.6.4 Loop Skewing ............................................................ 66
  2.6.5 Other Applications ..................................................... 70
2.7 Summary .......................................................................... 72

3 Compiling for Short Vector Processors .................................. 76
  3.1 Introduction ................................................................... 77
  3.2 Related Work .................................................................. 79
  3.3 Scalarization Strategy for Short Vector Processors .......... 81
  3.4 Data Alignment on Short Vector Processors .................... 84
  3.5 Experiments .................................................................. 90
3.5.1 SSE on Pentium 4 .............................................. 91
3.5.2 Altivec on PowerPC G4/G5 ............................. 95
3.6 Summary ......................................................... 99

4 Compiling for CELL Processor ............................................. 101

4.1 Introduction ....................................................... 102
4.1.1 CELL Processor ............................................. 103
4.1.2 Code Generation for CELL Processor ................. 105
4.1.3 Scalarization on CELL Processor ...................... 106
4.2 Related Work .................................................... 107
4.3 Dependence-based Code Generation ....................... 109
4.3.1 Parallelization and Vectorization .................... 109
4.3.2 Data Movement and Alignment ....................... 115
4.4 Scalarization and Temporary Array Allocation ........ 118
4.4.1 Allocation Loop Equals Parallelization Loop ........ 119
4.4.2 Allocation Loop Inside Parallelization Loop ....... 121
4.4.3 Allocation Loop Outside Parallelization Loop .... 122
4.5 Experiments ..................................................... 123
4.6 Summary ........................................................ 131

5 Performance Tuning of Loop Fusion ................................... 132

5.1 Introduction ..................................................... 132
5.2 Related Work ................................................... 134
5.3 Empirical Tuning of Loop Fusion ......................... 135
5.3.1 Parameterization of Loop Fusion ................. 137
5.3.2 Configuration Space ................................. 139
5.3.3 Code Generation ..................................... 141
5.3.4 Directly Executing Parameterized Code ............. 142
5.4 Experiments ........................................... 142
5.4.1 Experimental Setup .................................. 143
5.4.2 Parameterized vs. Conventional Output .............. 145
5.4.3 Properties of the Fusion Configuration Space ....... 148
5.5 Summary .............................................. 150

6 Conclusions ............................................ 152
6.1 Contributions ......................................... 153
6.2 Future Work .......................................... 155
List of Figures

1.1 Naive scalarization, failsafe scalarization ........................................ 3
1.2 Data reuse across array statements .................................................... 10

2.1 Naively scalarized code, dependence matrix, scalarization dependence matrix ................................................................. 15
2.2 Loop alignment example (1) ................................................................. 21
2.3 Loop alignment example (2) ................................................................. 21
2.4 Loop alignment example (3) ................................................................. 22
2.5 Loop alignment example (4): removing alignment section .................. 22
2.6 Variations of loop alignment ............................................................... 27
2.7 Loop skewing example: naively scalarized code and scalarization dependence matrix ................................................................. 29
2.8 Loop skewing example: $j = 2I + 3J$ ..................................................... 29
2.9 Loop skewing example: move j outside and updated dependence matrix 30
2.10 Loop skewing formula search algorithm ............................................. 31
2.11 Scalarization algorithm ................................................................. 33
2.12 Stencil diagram ................................................................. 35
2.13 $O(1)$ case examples: divisible stencils ........................................... 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.14</td>
<td>Algorithm for divisible stencil: O(1) temporary storage</td>
<td>38</td>
</tr>
<tr>
<td>2.15</td>
<td>O(n) case examples: indivisible stencils</td>
<td>39</td>
</tr>
<tr>
<td>2.16</td>
<td>Algorithm for indivisible stencil: O(N) temporary storage</td>
<td>39</td>
</tr>
<tr>
<td>2.17</td>
<td>O(n) proof: visited frontier</td>
<td>40</td>
</tr>
<tr>
<td>2.18</td>
<td>Stencil (b) case</td>
<td>41</td>
</tr>
<tr>
<td>2.19</td>
<td>Stencil (c),(d),(e) cases, value-out graph</td>
<td>41</td>
</tr>
<tr>
<td>2.20</td>
<td>Property of minimal indivisible stencil</td>
<td>47</td>
</tr>
<tr>
<td>2.21</td>
<td>v-Compression</td>
<td>49</td>
</tr>
<tr>
<td>2.22</td>
<td>O(1) case: naively scalarized code</td>
<td>51</td>
</tr>
<tr>
<td>2.23</td>
<td>O(1) case: intermediate and final scalarization dependence matrices</td>
<td>51</td>
</tr>
<tr>
<td>2.24</td>
<td>O(1) case: final scalarized code</td>
<td>52</td>
</tr>
<tr>
<td>2.25</td>
<td>Full 2D stencil scalarized with loop alignment</td>
<td>53</td>
</tr>
<tr>
<td>2.26</td>
<td>Full 2D stencil scalarized with lazy loop alignment</td>
<td>54</td>
</tr>
<tr>
<td>2.27</td>
<td>Full 2D stencil scalarized with reduced lazy loop alignment</td>
<td>55</td>
</tr>
<tr>
<td>2.28</td>
<td>Performance data on SGI (normalized to failsafe version, nine-point stencil)</td>
<td>57</td>
</tr>
<tr>
<td>2.29</td>
<td>Commercial compilers, running time (normalized to original version, nine-point stencil)</td>
<td>59</td>
</tr>
<tr>
<td>2.30</td>
<td>Commercial compilers, running time (normalized to original version, five-point stencil)</td>
<td>61</td>
</tr>
<tr>
<td>2.31</td>
<td>Fine tuning for backend compiler: loop unrolling (normalized to align version, nine-point stencil)</td>
<td>62</td>
</tr>
<tr>
<td>2.32</td>
<td>Full 2D stencil scalarized with reduced lazy loop alignment, rotating array indices and loop distribution</td>
<td>63</td>
</tr>
<tr>
<td>2.33</td>
<td>Fine tuning for Itanium 2: loop distribution (nine-point stencil)</td>
<td>64</td>
</tr>
</tbody>
</table>
2.34 Using vectorizer in Intel compiler for Pentium 4  
2.35 Loop skewing (3D), running time (normalized to original version)  
2.36 Loop skewing (2D), running time (normalized to original version)  
2.37 Loop skewing (3D), N=K=200, varying size of the innermost dimension (M)  
2.38 Loop skewing (3D), N=50, K=800, varying size of the innermost dimension (M)  
2.39 Loop skewing (3D), N=800, K=50, varying size of the innermost dimension (M)  
3.1 Performance penalty for unaligned data accesses  
3.2 Simple scalarization example  
3.3 Scalarization example  
3.4 Vectorized scalarization example  
3.5 Scalarization example (g): vector register allocated  
3.6 Scalarization example (h): vectorized scalar replacement and data alignment  
3.7 Scalarization algorithm considering data alignment  
3.8 Running time of 3point and 5point on Pentium 4  
3.9 Running time of 3point and 5point on PowerPC G5  
4.1 Architecture of CELL processor  
4.2 Fork-and-join execution model and compilation model on CELL  
4.3 Example of a loop nest and its dependences  
4.4 Cellization analysis algorithm  
4.5 Parallelizing a loop with anti-dependence
4.6 Cellization code generation algorithm ........................................ 113
4.7 Example of code generation .................................................. 114
4.8 Barrier and uni-directional synchronization using mailboxes .......... 115
4.9 Multi-buffering analysis and allocation algorithm ....................... 116
4.10 DMA transferring unaligned data .......................................... 117
4.11 Failsafe method for cellization .............................................. 119
4.12 Loop alignment method for cellization ..................................... 120
4.13 Pseudo SPE code for temporary allocation in the innermost loop ... 121
4.14 Parallelizing an array statement with anti-dependence ............... 121
4.15 Reduced lazy loop alignment method for cellization .................. 123
4.16 Testing codes ..................................................................... 125
4.17 Performance of 1D (LN1) and 2D (LN2) stencils ....................... 126
4.18 Performance of loop (ANTI) with anti-dependence .................... 126
4.19 Performance of swim and mgrid .......................................... 126
4.20 Performance for swim and mgrid before and after thread cost reduction 130

5.1 Empirical tuning of loop fusion ............................................. 137
5.2 Parameterization of loop fusion ............................................. 138
5.3 Number of fusion configurations ............................................. 140
5.4 Correlation of cycles on Intel Xeon ......................................... 147
5.5 Ratio of measurements on Intel Xeon: parameterized vs clean version 147
5.6 Speedup vs. register pressure on Xeon (left) and UltraSPARC (right) 150
List of Tables

2.1 Summary of scalarization transformations ........................................... 32
2.2 Hardware counter readings for grid size 800 × 800 on Itanium 2 .......... 62
2.3 Speedup after turning vectorizer on inside Intel compiler for Pentium 4 64
2.4 A spectrum of versions of code for tuning ........................................... 66
3.1 Versions of the programs ................................................................. 91
3.2 Running time of 3point on Pentium 4 .............................................. 92
3.3 Running time of 3point (C implementation) on Intel Pentium 4 ........ 93
3.4 Running time of 5point (C implementation) on Intel Pentium 4 ........ 93
3.5 Vector load and store instructions of 3point (C implementation) on
   Intel Pentium 4 ................................................................................. 94
3.6 Running time of 3point on Apple PowerPC G4 and G5 ................. 96
3.7 Running time of 3point (C implementation) on Apple PowerPC G4
   and G5 ......................................................................................... 97
3.8 Running time of 5point (C implementation) on Apple PowerPC G4
   and G5 ......................................................................................... 98
4.1 CELLizer options .......................................................................... 124
4.2 SPE thread creation cost ................................................................. 129
4.3 Running time of *ANTITMP* before and after thread cost reduction . 129
Chapter 1

Introduction

Array syntax was introduced in APL [28] and later supported in many other languages such as MATLAB [39], Fortran 90 [1, 10], and ZPL [17], and language extensions such as the Overture C++ library [11]. It adds expressive power to a language by allowing operations on and assignments to whole arrays and array sections. Thus, array syntax makes it possible for scientific programmers to write simple and concise array code.

However, state-of-the-art vendor compilers have failed to map array syntax efficiently to the underlying architectures. In the following, we introduce array syntax, its performance problems, and the solutions developed in this dissertation that significantly improve the performance of programs written in array syntax.

1.1 Array Syntax: Syntax, Semantics and Implementation

In many languages, array syntax uses subscript triplets to specify all or part of an array to be involved in a computational operation. For example, in Fortran 90, the
components of the triplet, from left to right, specify lower bound, upper bound, and stride. If the stride component is omitted, it is presumed to be 1. As an example, the array assignment

\[ A(1:4) = B(2:5) + C(2:5) \]

adds elements of \( B \) and \( C \) from 2 to 5 element-wise and stores the results into \( A \) at location from 1 to 4. Array syntax can be used in array assignments, conditional array (WHERE) statements, WHERE constructs, and intrinsic functions.

Implementing array statements on a given target machine architecture is the responsibility of the compiler for that machine. When the machine is a scalar processor or a vector processor whose vector length is smaller than the size of the array section, the array statement must be converted into a loop that maintains the correct semantics, by a process called scalarization.

The simplest scalarization, naive scalarization, iterates through all triplets in the subscript. For the example statement in the previous paragraph, the naively scalarized code is:

```
DO I = 1, 4
   A(I) = B(I+1) + C(I+1)
ENDDO
```

However, scalarization must preserve the “fetch before store” semantics of array assignments; i.e., all source elements in the computation must have original values. Under these semantics, the naive scalarization shown in Figure 1.1 (a) is incorrect for the statement

\[ A(2:5, 2:5) = A(2:5, 1:4) + A(3:6, 2:5) \]
DO J = 1, 4
DO I = 1, 4
A(I+1, J+1) = A(I+1, J) + A(I+2, J+1)
ENDDO
ENDDO

DO J = 1, 4
DO I = 1, 4
T(I, J) = A(I+1, J) + A(I+2, J+1)
ENDDO
ENDDO

(a) (b)

Figure 1.1: Naive scalarization, failsafe scalarization

because the naively scalarized loop has different meaning—the values of A(I+1, J) loaded after the first J iteration are computed on previous J iterations.

To preserve the correct semantics, many compilers allocate a full size temporary array and split the assignment into two loops [55, 45]. The advantage of this failsafe scalarization method is that it is easy to implement and always succeeds in generating correct code. Figure 1.1 (b) shows the generated code for the above example using this method. The disadvantage is that the full size temporary, which is written in the first loop and read in the second loop, increases the memory footprint of the application and may cause excessive cache misses. This disadvantage has translated to performance problems observed by application developers [45].

A natural question to ask is: Is it possible to write a program in array syntax and still achieve high performance? The answer to this question is important because otherwise array syntax, though a high-level language feature in many languages, may not be widely used by application developers. Therefore, we present in this dissertation compilation strategies that efficiently compile array syntax for high performance.
1.2 Thesis

This dissertation illustrates that the performance problems for programs written in array syntax stem from the lack of effective compilation strategies from current state-of-the-art vendor compilers to solve the following problems: (1) reducing the size of allocated temporary array; (2) extending solutions to the evolving architectures; (3) applying loop fusion to multiple array statements.

This dissertation develops automatic compilation strategies that improves the performance of array statements upon current state-of-the-art implementation across a variety of architectures. Implementing such strategies in a research compiler demonstrates that significant improvements can be achieved by improving memory hierarchy utilization through advanced scalarization techniques, exploiting parallelism in vector and multi-core processors, and exploring the properties of loop fusion space for empirical performance tuning.

In the rest of this chapter, Section 1.3 explains the issues that affect the performance of scalarized code for both a single array statement and a collection of array statements. In particular, three types of architectures are discussed. The solutions in our compilation strategy for these issues are briefly introduced. Section 1.4 gives the structure of the rest of this dissertation, which will present these issues and our solutions in detail.

1.3 Compiling Array Syntax for High Performance

Since executing an array statement strictly following the expression requires a vector machine whose length is longer than or equal to that of array sections, a scalarization
strategy must be in place to map array sections to the available machine resources on various architectures, which only support vector operations of limited length (scalar processors can be considered as vector machines of length one). Thus, the first and foremost responsibility of scalarization is to preserve the semantics of the original array statements. However, as pointed out in the previous section, obtaining high performance for the scalarized code is also important.

In this section, we introduce two contributing factors that affect the performance of the scalarized code, which will in turn affect the development of an effective scalarization strategy: 1) the characteristics of the underlying architecture, and 2) the inter-statement data reuse.

1.3.1 Architectures

What we observe as a trend today is that microprocessors have evolved from scalar machines into machines with increasing parallelism on the same chip. This trend represents an alternative way to increase performance per chip instead of scaling up a chip’s operating frequency, which has begun to cause heat dissipation problems. Parallelism is introduced at two levels of granularity. At a fine granularity, vector function units are added so that more data items can be processed per vector instruction; at a coarse granularity, more computation cores are added so that multiple computation threads can be executed simultaneously. Therefore, in this dissertation, we will address not only the performance problems related to the traditional scalar processors, but also those related to the parallelism on emerging microprocessors.
Scalar Processors

We use the term scalar processors to include the microprocessors that have a scalar instruction set; i.e., an operand of an instruction is a single data item, be it an integer, a floating point, etc. Almost all microprocessors support such a scalar instruction set.

On scalar processors, a memory hierarchy that consists of registers and multi-level caches of limited sizes is usually placed on-chip to reduce the latency of off-chip memory accesses. Since the on-chip registers and cache are much faster than the memory accesses, the performance of an application can be improved if the on-chip memory hierarchy is put into good use. As shown in Figure 1.1 (b), the temporary array allocated by the failsafe scalarization requires more memory accesses and induces a larger memory footprint that may cause excessive cache misses. Thus, an effective scalarization strategy on scalar processors should find a solution to avoid the use or reduce the size of the temporary array.

Solution We present a novel application of two well-known compiler strategies—loop alignment and loop skewing—to address this problem [61]. To our knowledge, neither of these transformations has been previously used in scalarization algorithms. We prove that a combination of loop alignment and loop skewing can achieve an allocated temporary array whose size is asymptotically minimum for stencil computations written in array syntax. A scalarization compiler has been developed to implement these two compiler strategies in addition to previous methods proposed by other researchers.
Short Vector Processors

By short vector processors, we refer to the microprocessors that have a short vector instruction set; i.e., an operand of an instruction consists of several data items. Examples of short vector processors are the x86 and x86-64 processors from Intel and AMD with SSE instruction set, and the PowerPC processors from Motorola and IBM with Altivec instruction set. Note that a short vector processor can have a scalar instruction set as well, and the scalar operations can be managed to operate using the vector instructions. Therefore, short vector processors can be considered as a super set of scalar processors. For example, to an application, a x86 processor with SSE can still be viewed as a scalar processor if none of the vector instructions is used.

Compared to conventional vector machines, the vector length on short vector processors is usually short (16 bytes), and there is no support for strided vector data accesses; i.e., vector instructions can only access data in continuous memory. Nevertheless, when fully utilized, these vector instructions can speed up an application by a maximal factor of the vector length for the corresponding data type. Thus, when scalarizing array statements on these processors, we should generate code targeting these vector instruction sets, by a process called vectorization.

Short vector processors have a memory hierarchy similar to the scalar processors. However, with the addition of the vector instruction sets, there is a restriction that will affect the effectiveness of vectorization: Vector data accesses to aligned data on vector boundaries (16-byte boundaries for both SSE and Altivec) are faster than those to unaligned ones. Thus it is desirable to have as many memory accesses aligned as possible in scalarized code.

Solution We demonstrate that loop alignment as an iteration space transfor-
mation can help satisfying the data alignment constraint [62]. When used together with previous methods as a complement, loop alignment also reduces register pressure and in turn further improves the performance on short vector units. Moreover, we integrate this data alignment strategy into our scalarization algorithm so that it reduces the allocated temporary array size and improves data alignment performance at the same time. The scalarization compiler is extended with an automatic vectorization implementation for both SSE and AltiVec. For general loop nests that are vectorizable, this compiler vectorizes them as well because they can be considered as intermediate results of scalarization on a scalar processor.

**Multi-Core Processors**

In addition to vector capabilities, as a trend for current microprocessor development, on-chip parallelism is also introduced at core level to produce multi-core processors. Examples are the dual core processors released by Intel and AMD, and the CELL processor developed by Sony, Toshiba and IBM. For Intel and AMD multi-core chips, each core is identical and the chips are called *homogeneous*. The CELL processor has 1 PPE (PowerPC) core and 8 identical SPE cores (other than PowerPC) and is called *heterogeneous*. On these example processors, each core is a short vector processor. When an application can be divided into sub-tasks and the sub-tasks can run on the cores simultaneously, the running time of an application could be shortened compared to running on a single core. This application subtasking process is called *parallelization*. On such multi-core chips, for maximum performance, scalarization should have a parallelization strategy as well, in addition to the one for vectorization.

Memory hierarchy becomes more complicated on multi-core chips compared to single-core chips. The homogeneous processors mentioned above have either a shared
cache or a separate cache for each core. On the CELL processor, each SPE core has its own 256KB local store memory (LS), while PPE core has a normal two level cache and is responsible for scheduling computation tasks onto SPEs. Computation on a SPE core can only access LS and the data transfer between LS and the off-chip main memory is explicitly controlled by direct memory accesses (DMA).

We choose to focus on the CELL processor because of its heterogeneity and the explicitly controlled local store memory. Similar to Altivec, the vector instructions on a SPE also require proper data alignment. Moreover, the performance of DMA data movements is the most efficient on certain data alignment boundaries, thus data alignment should also be considered for DMA data transfers.

**Solution** Since other issues related to full utilization of the SPEs such as instruction scheduling and branch prediction are handled by the backend compiler, we focus on the optimizations for improving data alignment and data movement performance, as well as reducing the size of allocated temporary array during the scalarization process. We show that loop alignment can help both vector and DMA data alignment, and that temporary allocation reduction for scalarization does not prohibit effective parallelization and vectorization. The scalarization compiler for the short vector processors is extended with automatic parallelization for PPE and SPEs, support for the SPE’s vector instruction set, multi-buffering data movement, and automatic synchronization generation [63]. Our compiler can also generate code for general loop nests that can be parallelized and vectorized for a CELL processor.
1.3.2 Inter-Statement Data Reuse

Having explained performance problems and compilation strategies for single array statements, we now shift the discussion to those for multiple array statements.

The expressiveness of array syntax is limited to each array statement. Certain compiler transformations including loop fusion target a larger scope than a single statement to improve an application’s performance. Figure 1.2 shows such an example for improving data reuse across multiple statements. If \( N \) is very large and each array statement is considered separately, array \( A \) and \( B \) will have to fetch from memory again when the second array statement is executed. However, if the two scalarized loops are fused together as shown in Figure 1.2 (b), subsequent uses of \( A \) and \( B \) by the second statement will get the value from the registers or cache instead of accessing the memory again. Therefore, a loop fusion transformation based on data reuse after scalarization is useful for improving memory hierarchy performance across different array statements.

\[
\begin{align*}
C(1:N) &= A(1:N) + B(1:N) \\
D(1:N) &= A(1:N) - B(1:N)
\end{align*}
\]

\[
\text{DO } I = 1, N \\
C(I) &= A(I) + B(I) \\
D(I) &= A(I) - B(I) \\
\text{ENDDO}
\]

Figure 1.2: Data reuse across array statements

The difficulty of loop fusion is to statically select proper sets of loops to fuse, given that there are many different fusion configurations to choose from. Though many vendor compilers implement loop fusion as a standard optimization for memory hierarchy performance, they use heuristics to decide the fusion configurations instead of examining all possible ones because of time constraints. Recent research has resorted to empirical performance tuning to determine the best configuration of
compiler optimizations such as loop tiling and unroll-and-jam on a specific architecture [54, 32, 25, 15]. However, none of previous research has formalized parameterization of loop fusion or explored the properties of the loop fusion space. The recent loop fusion tuning work in [46] searches through a relatively small space of pair-wise loop fusion rather than through the entire space of reordering loop fusion.

Solution We have developed a strategy to parameterize loop fusion such that the entire configuration space of reordering loop fusion can be represented by a set of integer parameters [64]. The parameterization process produces a single parameterized version of the original program, and it outputs dependence graphs and other analysis information such as register pressure estimates. Our empirical tuning engine can then generate and evaluate all legal configurations and identify the best choice.

However, the space of the reordering loop fusion is usually very large and hence it is very expensive (if not infeasible) to evaluate the whole space to find the optimal configuration. To obtain insights into how to design effective searching techniques and search space pruning heuristics, we conducted a study to explore the properties of the loop fusion space. Our results on the space of pair-wise fusion identify that Gray-ordering provides a good coordinate for fast searching methods such as hill-climbing with random restart, and that resource constraints such as register pressure estimates supply effective search space pruning criteria. We also illustrate that it is feasible to use the parameterized performance tuning to speed up the search process.

1.3.3 Summary

In summary, this dissertation addresses two important problems related to compiling languages with array syntax for high performance: (1) scalarize a single array
statement into a loop with consideration of memory system constraints, whether the underlying architecture be a scalar processor, a short vector processor, or a multi-core processor with vector capability and explicitly managed local scratch pad memory, and (2) on the resulting loops after the scalarization of multiple array statements, evaluate the effectiveness of loop fusion to further improve inter-statement data reuse in an automatic empirical performance tuning framework.

1.4 Structure of Dissertation

In what follows, to reduce memory traffic and improve cache utilization, this dissertation first develops a novel strategy that minimizes the allocated temporary arrays using loop alignment and loop skewing on scalar processors in Chapter 2. It then extends the minimization strategy to exploit the increasing on-chip parallelism on evolving architectures that offer vector (e.g., SSE and AltiVec) and multi-core (e.g., CELL) capabilities in Chapter 3 and Chapter 4. In addition, new techniques are employed to boost performance by improving data alignment and managing data movement, both important on these new architectures. In Chapter 5, this dissertation parameterizes loop fusion for performance tuning and explores the properties of the space of all possible loop fusion configurations to expedite performance tuning of loop fusion for increasing data reuse across multiple array statements. In each chapter, related work is reviewed and our experimental results are presented. Finally, Chapter 6 gives the conclusions and summarizes the five contributions.
Chapter 2

Compiling for Scalar Processors

In this chapter, we first review the scalarization problem for scalar processors in Section 2.1, that the temporary array allocated by a failsafe scalarization method poses problems in the memory hierarchy, when a naive scalarization does not preserve the semantics of the original array statements. To determine whether a naively scalarized result is correct, we introduce a scalarization dependence matrix, which also provides information for guiding various optimizations that are designed to avoid or reduce the use of temporary arrays. After reviewing related work in Section 2.2, we present a novel application of two optimizations, loop alignment and loop skewing, for scalarization in Section 2.3 and Section 2.4. Integrating these two optimizations with existing techniques, we apply a combined scalarization strategy to stencil computations and prove that the asymptotically minimal memory allocation can be achieved in Section 2.5. Our experiments with loop alignment and loop skewing presented in Section 2.6 demonstrate that it is extremely effective in improving memory hierarchy performance of Fortran 90 array code on standard scalar processors. Though our experiments are conducted on Fortran 90 codes, the strategies we developed here should
apply to other array languages with similar semantics. Section 2.7 summarizes this chapter.

2.1 Introduction

2.1.1 Scalarization Problem on Scalar Processors

In Chapter 1 we mentioned that a semantic execution of an array statement requires a vector machine that supports arbitrary vector length. Since a scalar processor can only support a vector length of 1, the array statement must be converted into a loop that maintains the correct semantics, when compiled onto a scalar processor by a process called scalarization.

Scalarization presents a significant technical problem because of the difference in semantics between an array assignment and a sequential loop—the array assignment must be implemented as if all inputs are fetched before any outputs are stored. Because a loop intermixes loads and stores, it is often necessary for a compiler to use a temporary array to save the values to be stored and then copy these values back to the array being assigned in a separate loop [55]. This failsafe method always produces correct result. However, the temporary arrays allocated by the failsafe method represent extra storage and memory accesses and can cause performance problems in the memory hierarchy.

Recall the example in Chapter 1 that illustrates a naive scalarization does not preserve the correct semantics for array statement

\[ A(2:5, 2:5) = A(2:5, 1:4) + A(3:6, 2:6) \]

because the naively scalarized loop has different meaning—the values of \( A(I+1, J) \)
loaded after the first J iteration are computed on previous J iterations, as shown in Figure 2.1 (a).

\[
\begin{array}{c}
\text{DO } J = 1, 4 \\
\text{DO } I = 1, 4 \\
A(I+1, J+1) = A(I+1, J) + A(I+2, J+1)
\end{array}
\]

(a)

\[
\begin{array}{ccc}
J & I & \text{type} \\
< & = & \text{true} \\
= & < & \text{anti} \\
= & > & \text{anti}
\end{array}
\]

(b)

\[
\begin{array}{ccc}
J & I & \text{type} \\
< & = & \text{true} \\
= & < & \text{anti} \\
= & > & \text{anti}
\end{array}
\]

(c)

Figure 2.1: Naively scalarized code, dependence matrix, scalarization dependence matrix

### 2.1.2 Scalarization Dependence Matrix

To determine whether a naively scalarized loop nest preserves the semantics of the original array statement, Allen and Kennedy [4] have observed that the scalarized code is correct only if the generated loop nest carries no true dependence. We now briefly introduce the concept of data dependences [4]. Two array references have a data dependence between them if they access to the same memory location during the same execution. Simply put, the true, anti-, output and input dependences correspond to read after write, write after read, write after write, and read after read references to the same memory location, respectively. For data dependences between two array references in the same loop nest, they are classified as loop-independent and loop-carried, depending on whether the accesses to the same memory location occur in the same loop iteration or in difference loop iterations, respectively.

Allen and Kennedy have defined the dependence matrix for a loop nest to be a matrix in which each row is a dependence vector for some dependence in the nest and every such direction vector is included as a row. Note that nothing is lost if there is only one row per distinct direction vector—that is no row is duplicated. The
columns are the loops from the outermost to the innermost. Each entry in a direction vector is a direction symbol with an integer distance value that indicates the relative timing of two references accessing the same memory location during the iterations of the corresponding loop. Note that an integer distance value is not always available. A dependence is said to be carried by a loop if the corresponding entry is the first non-"=" entry in the direction vector.

For instance, in the example code shown in Figure 2.1 (a), there is a true dependence from A(I+1, J+1) to A(I+1, J) carried by loop J, and an antidependence from A(I+2, J+1) to A(I+1, J+1) carried by loop I, the corresponding dependence matrix for this loop nest is shown in Figure 2.1 (b).

In this chapter, all algorithms are based on the scalarization dependence matrix, a modified form of the dependence matrix for the naively scalarized loop nest. Note that, in nests generated from Fortran 90 assignments, there cannot be any carried output dependences. Since there are only loop-carried true dependences and antidependences, the type information associated with each dependence vector in the ordinary dependence matrix is omitted. In this form, we distinguish true dependences from antidependences by whether the leftmost non-"=" direction is "<" (for true dependence) or ">" (for antidependence), as shown in Figure 2.1 (c). Note that this is different from the standard dependence matrix, in which every dependence must either be all "=" or have a leading "<".

For a scalarization to be correct, the scalarized loop nest cannot carry any true dependences. Therefore, in the corresponding scalarization dependence matrix, the first non-"=" entries of all dependence vectors must be ">", such a matrix is called valid scalarization dependence matrix. In our approach, we consider the dependences caused by the original left-hand-side array only, excluding dependences introduced
by the temporary arrays the algorithms may generate.

2.2 Related Work

As we indicated earlier, array assignment in Fortran 90 must be implemented to conform to "fetch-before-store" semantics—that is, all array elements referenced on the right hand side of the assignment should appear to be fetched before any stores occur. However, no constraint is placed on the order in which array elements on the right hand side must be fetched; i.e., it doesn’t matter which dimension is computed first, or which element within a dimension is computed first.

This observation lays the foundation for all scalarization transformations that try to eliminate or reduce the use of temporary arrays. By manipulating the iteration space, these transformations adjust the evaluation order of array elements, and hopefully eliminate the loop-carried true dependences, which at the same time transform the original scalarization dependence matrix into a valid one.

We now briefly describe four existing strategies that have been used to ensure a correct scalarization. Three of these also reduce the storage requirements of temporary array storage.

The *failsafe method* allocates a full size temporary array and splits the assignment into two loops [55]. The first loop performs the computation, while the second loop stores the result. The advantage of this approach is that it is easy and always succeeds in generating correct code. The disadvantage is that the full size temporary, which is iterated over in each loop, may cause excessive cache misses. For SMP machines, on the other hand, this method always allows parallelization of the generated loop [48].

*Loop reversal* runs a loop backwards, transforming all true dependences carried
by that loop into antidependences [53]. As an example, consider the statement

\[ A(2:5) = A(1:4) + C \]

which, after naive scalarization, becomes:

```plaintext
do i = 1, 4
    a(i+1) = a(i) + c
endo
```

Although this loop carries a true dependence, the problem can be eliminated by running the loop backward, as follows:

```plaintext
do i = 4, 1, -1
    a(i+1) = a(i) + c
endo
```

which has the same meaning as the original array assignment. Although it is successful in this case, loop reversal will also turn a loop-carried antidependence into true dependence; thus, it cannot handle the situation when both true dependences and antidependences are carried by the same loop.

*Loop interchange* [2] moves scalarization loops that have only ">" entries in the scalarization dependence matrix to outer positions within a nest, which may change a loop-carried true dependence into a loop-carried antidependence. For example, The array assignment

\[ M(2:5, 2:5) = M(3:6, 1:4) + M(2:5, 3:6) \]

has the naive scalarization loop

```plaintext
do j = 1, 4
do i = 1, 4
    m(i+1, j+1) = m(i+2, j) + m(i+1, j+2)
endo
endo
```
The scalarization dependence matrix for this loop is
\[
\begin{bmatrix}
  J & I \\
  <_1 & >_{-1} \\
  >_{-1} & =_0
\end{bmatrix}
\]

Upon examination of this matrix, it is easy to see that interchanging the J and I loops will provide a correct scalarization.

*Input prefetching* [4] gets an old array value one or more iterations before it is needed and saves it, so that when a left hand side element is written, its old value has already been copied into a saved location, preserving the correctness of the computation. As an illustration, consider the statement

\[ A(2:5) = A(1:4) + A(3:6) \]

The naive scalarization

\[
\begin{array}{l}
  \text{DO } I = 1, 4 \\
  \quad A(I+1) = A(I) + A(I+2) \\
  \text{ENDDO}
\end{array}
\]

cannot be fixed by loop reversal. However, we can restore correctness by prefetching the \( A(I) \) as follows:

\[
\begin{array}{l}
  \text{TO} = A(1) \\
  \text{DO } I = 1, 4 \\
  \quad T1 = A(I+1) \\
  \quad A(I+1) = \text{TO} + A(I+2) \\
  \quad \text{TO} = T1 \\
  \text{ENDDO}
\end{array}
\]

The cost is two extra memory locations, which is much better than the failsafe method which requires a full-sized vector temporary.
Putting these techniques together, a standard scalarization algorithm would iterate from the outermost loop to the innermost loop, ensuring that each loop carries only antidependences by applying loop reversal, loop interchange, input prefetching, and the failsafe method, in that order. In the following we show how loop alignment and loop skewing can be used to improve this standard approach to scalarization.

2.3 Loop Alignment

Loop alignment [4, 43] is an iteration space transformation technique that can be used to eliminate loop-carried true dependences and thus to preserve the array assignment semantics during scalarization. In what follows, we describe how loop alignment is applied to scalarization, when it can be used, its profitability, and how it compares to other techniques. We also show two different variations of loop alignment.

2.3.1 Loop Alignment in Scalarization

We begin by presenting an example to show how loop alignment can be applied during scalarization of the statement: $A(2:11) \times A(1:10) \times A(3:12)$. Figure 2.2 shows a naively scalarized version that is wrong because the $1$ loop carries a true dependence (a failsafe version is shown on the right).

The two loops in the failsafe version cannot be fused together because the resulting loop would carry a backward true dependence of distance $1$. However, if the two loops are aligned by this distance, the loop-carried backward true dependence will be converted to a loop-independent dependence and the two loops can be safely fused, as shown in Figure 2.3.

After the loops are fused, we see that we don’t really need a temporary array of
size 10, size 2 is ample, as shown in Figure 2.4. Note that the scalarization dependence matrix becomes valid after the transformation.

Though the whole process is shown in four steps above, we can easily merge them into a single transformation, which we call loop alignment, simply because the alignment distance and the temporary array size can be determined at the very beginning. Notice that the final code shape consists of three parts: a computation section, a store section and an alignment section, as shown in the above example. The alignment section can often be eliminated either by loop unrolling or by using rotating array indices as shown in Figure 2.5.
ALLOCATE T(2)
T(1) = A(1) + A(3)
DO I = 1, 9
   T(2) = A(I+1) + A(I+3) ! computation section
   A(I+1) = T(1) ! store section
   T(1) = T(2) ! alignment section
ENDDO
A(11) = T(1)
DEALLOCATE T

(e) temporary size reduced  (f) scalarization dependence matrix

Figure 2.4: Loop alignment example (3)

DO I = 1, 6, 2
   T(2) = A(I+1) + A(I+3)
   A(I+1) = T(1)
   T(1) = A(I+2) + A(I+4)
   A(I+2) = T(2)
ENDDO
DO I = 9, 9
   T(2) = A(I+1) + A(I+3)
   A(I+1) = T(1)
   T(1) = T(2)
   T(MOD(I, 2) +1) = A(I+1) + A(I+3)
   A(I+1) = T(MOD(I-1, 2) +1)
ENDDO

(g) loop unrolling  (h) rotating array indices

Figure 2.5: Loop alignment example (4): removing alignment section

2.3.2 Applicability and Profitability of Loop Alignment

Since loop reversal doesn’t introduce any temporary arrays when a loop carries only true dependences, loop alignment should only be applied when a loop carries both true dependences and antidependsences. For alignment to be possible, the distances of these true dependences has to be known at compile time. In practice, we only align dependences with small constant distances.

To apply loop alignment, first analyze the scalarization dependence matrix to find a candidate. A loop is alignable if the corresponding column in the dependence matrix satisfies the following two conditions:

1. There are one or more “<”s and one or more “>”s, zero or more “=”s or “>=”s,
but no other direction symbols.

2. The distances corresponding to all "<" entries' dependence distances are known at compile time and the maximum of these distances is no greater than some small threshold. We use 3 for the threshold in our experiments. The maximum of the distances is the alignment distance.

However, within a loop nest, there could be more than one alignable candidate. For a given candidate, let the alignment distance be $d$, its loop size be $N$ and the iteration volume of all loops including the candidate be $S$, we will have to allocate a temporary array of size $(d+1)*S/N$. Therefore, we choose the candidate with the minimum value for this expression and move it to the outermost position. Note that loop alignment can only reduce the size of temporary array in one dimension. After it is allocated, subsequent loop alignments have no effect in reducing the temporary size.

Once the alignment is performed, the entries in the column corresponding to the aligned loop need to be updated:

1. Keep "->" entries unchanged.

2. Change "->" and "=" entries to "->".

3. For "<" entries, if the entry's dependence distance is equal to (less than) the alignment distance, change it to "=" (">"), respectively.

Essentially, we are subtracting $d$ from the dependence distances of all entries in the aligned column of the scalarization dependence matrix. After updating the entries, all dependences with a "->" in this column can be removed from the matrix since they are now antidependences carried by the aligned loop. When the algorithm moves to the next (inner) loop in the loop nest, these dependences need not be considered.
2.3.3 Loop Alignment vs. Input Prefetching

The input prefetching strategy, as described in the literature [4], is also intended to reduce the requirement for temporary storage when a scalarization loop carries both true dependences and antidependsences. Which method should we use, loop alignment or input prefetching? Based on the following analysis, we can see that loop alignment is more powerful than input prefetching.

1. For every "<" entry in the current column, input prefetching needs to be applied to correct it. However, one loop alignment will correct all these entries. It is possible that these multiple prefetches can be merged into one, but doing so requires the analysis of the shape of the array reference subscripts in inner dimensions, which makes the scalarization more complicated.

This can be shown in the following example:

\[
\]

Input prefetching would have to prefetch \(A(3:12, j-1:1)\) and \(A(12:21, j-2:1)\), while loop alignment only needs to temporarily hold the result \(A(11:20, j-2:1)\), where \(j\) is the current iteration.

2. Once loop alignment is applied, all "=" entries in the current column are turned into ">"s. Therefore, these dependences can be eliminated from consideration when proceeding to the inner loops. Input prefetching does not have this additional benefit.

This can be shown in the following example:

\[
\]
After input prefetching the first array reference on the right, the resulting scalarization dependence matrix is still not a valid one and thus needs further transformations, while one single loop alignment will make the resulting scalarization matrix a valid one.

2.3.4 Lazy Loop Alignment

Our scalarization algorithm has adopted a scheme that works from the outermost loop towards the innermost loop. In this scheme, loop alignment has been used to correct the current column only, as we have described above. As an alternative, we could increase the alignment distance by one, which would turn all the entries of the current column in the scalarization dependence matrix into ">"s. That would result in the current loop carrying the dependences that are not carried by some outer loop. Therefore, the scalarization can stop here, without going further to examine the remaining loops. We call this variation lazy loop alignment.

Because of the increased alignment distance, lazy loop alignment requires a larger temporary storage, extra peeled iterations, and extra register or array copying operations. Using the previous formula, we must now allocate a temporary array of size $(d+2) \times S/N$ instead of $(d+1) \times S/N$. In return, lazy loop alignment leads to a faster scalarization and a cleaner code since it doesn’t affect the code of inner loops, which are usually perfectly nested. It can also be applied to the case that the dependence directions and distances of the entries of the remaining columns are unknown at compile time.

The following example

$$A(2:11,2:11) = A(1:10, 1:10) + A(3:12, 1:10) + A(1:10, 3:12)$$
clearly illustrates the difference between loop alignment (Figure 2.6 (a)) and lazy loop alignment (Figure 2.6 (b)). Close examination of the code produced by lazy loop alignment reveals that the store section and part of the alignment section can actually be moved ahead of the computation section. This is because lazy loop alignment makes all entries of the current column in the scalarization dependence matrix ‘>’s, thus all antidependence are actually carried by the loop and not loop independent. With this code motion, we can see that the last array copying operation is redundant and therefore can be eliminated. This brings the size of temporary array back to \((d + 1) \times S/N\), as shown in Figure 2.6 (c). We call this variation reduced lazy loop alignment.

As our performance study in Section 2.6 will show, code generated by lazy loop alignment does not necessarily perform worse than those generated by loop alignment and reduced lazy loop alignment. The reason is that memory size is not the only contributing factor to the performance of scalarized code. The performance also depends on how well the backend compiler optimizes the scalarized code. Since the code generated by lazy loop alignment and reduced lazy loop alignment tend to be cleaner than that generated by loop alignment, the backend compiler may perform better analysis and optimizations such as dependence analysis and software pipelining, and thus generate better compiled code. The performance gained by the improved quality of backend compilation can compensate for the performance penalty caused by allocating slightly larger temporary memory. Similarly, backend compiler may schedule lazy loop alignment and reduced lazy loop alignment differently. However, since we don’t know exactly the tradeoff between the quality of backend compiled code and the memory footprint size, and we intend to treat the backend compiler as a black box, we may want to generate three versions of the code using loop alignment, lazy
(a) loop alignment

ALLOCATE T(10, 2)
DO I = 1, 10
   T(I, 1) = A(I, 1) + A(I+2, 1) + A(I, 3)
ENDDO
DO J = 1, 9
   T(I, 2) = A(I, J+1) + A(S, J+1) + A(I, J+3)
   T(I+1, 2) = A(I-1, J+1) + A(I-3, J+1)
   + A(I+1, J+3)
   A(I+1, J+1) = T(I, 1)
   T(I, 1) = T(I, 2)
ENDDO
A(I+1, J+1) = T(10, 1)
T(10, 1) = T(10, 2)
ENDDO
DO I = 1, 10
   A(I+1, 11) = T(I, 1)
ENDDO

(b) lazy loop alignment

ALLOCATE T(10, 3)
DO I = 1, 10
   T(I, 1) = A(I, 1) + A(I+2, 1) + A(I, 3)
   T(I, 2) = A(I, 2) + A(I+2, 2) + A(I, 4)
ENDDO
DO J = 1, 8
   DO I = 1, 10
      T(I, 3) = A(I, J+2) + A(I+2, J+2)
      + A(I, J+4)
      T(I, 1) = T(I, 2)
      T(I, 2) = T(I, 3)
   ENDDO
ENDDO
DO I = 1, 10
   A(I+1, 10) = T(I, 1)
   A(I+1, 11) = T(I, 2)
ENDDO

(c) reduced lazy loop alignment

ALLOCATE T(10, 2)
DO I = 1, 10
   T(I, 1) = A(I, 1) + A(I+2, 1) + A(I, 3)
   T(I, 2) = A(I, 2) + A(I+2, 2) + A(I, 4)
ENDDO
DO J = 1, 8
   DO I = 1, 10
      A(I+1, J+1) = T(I, 1)
      T(I, 1) = T(I, 2)
      T(I, 2) = A(I, J+2) + A(I+2, J+2) + A(I, J+4)
   ENDDO
ENDDO
DO I = 1, 10
   A(I+1, 10) = T(I, 1)
   A(I+1, 11) = T(I, 2)
ENDDO

! store section
! alignment section
! computation section

Figure 2.6: Variations of loop alignment

loop alignment and reduced lazy loop alignment, respectively. Then we use profiling tools to determine which version performs the best. Or with the assistance of run-time profiling tools such as UNIX system timer or PAPI [38], we could embed all versions in the generated code and perform the run-time adaptation to choose the best version.
2.4 Loop Skewing

Loop skewing is a transformation that has been used to remap an iteration space so that wavefront parallelism lines up directly with a loop iteration [4, 35, 56]. This transformation involves two or more loops that all carry dependences. By creating a new loop that is a linear combination of two or more loops, loop skewing tries to make one of the loops carry all the dependences when positioned in the outermost position so that the other loop can be parallelized. In what follows, we illustrate how loop skewing can help scalarization, when it can be applied and its profitability, and present a summary of scalarization transformations.

2.4.1 Loop Skewing in Scalarization

Before showing how loop skewing can help scalarization, we analyze how loop skewing manipulates the dependences in a nest. When associating two or more loops together and creating a new loop, subscripts of array references become coupled and thus a coupled subscript test such as the delta test [26, 4] needs to be performed. Take the case of two loops in which the loop indices are $I$ and $J$. Let the dependence matrix be $D$, the combination formula be $j = n \star I + m \star J$. The dependence entries for the new loop $j$ is $d_j = n \star d_i + m \star d_j$. This can be easily derived from the delta test. Let the array reference at the source and the sink be $f_1(I,J)$ and $f_2(I,J)$, initially we have $f_1(I, J) = f_2(I + \Delta I, J + \Delta J)$, now we have $f_1(I, \frac{i-n \star I}{m}) = f_2(I + \Delta I, \frac{j-n \star I + m \star \Delta I}{m})$, which gives us $\Delta J = \frac{n \star \Delta I}{m}$. Therefore, $\Delta j = n \star \Delta I + m \star \Delta J$.

Our goal for loop skewing in the context of scalarization is to obtain a new loop that carries only antidependences. In terms of the scalarization dependence matrix, we want to obtain a valid one. Let’s look at an example:

The naively scalarized code and the scalarization dependence matrix (with both directions and distances) are shown in Figure 2.7.

```plaintext
DO J = 1, N
   DO I = 1, N
      A(I+2, J+2) = A(I, J+4) + A(I+4, J+1) + A(I+2, J+4)
   ENDDO
ENDDO

\[
\begin{bmatrix}
  J & I \\
  >2 & <2 \\
  <1 & >2 \\
  >2 & =0
\end{bmatrix}
\]
```

Figure 2.7: Loop skewing example: naively scalarized code and scalarization dependence matrix

Now we need to determine what formula to use. Let the unknown formula be

\[j = n \ast I + m \ast J.\]

We know that in the new dependence matrix, \(\tilde{d}_j = n \ast \tilde{d}_I + m \ast \tilde{d}_J.\)

Therefore, we can resolve \(n\) and \(m\) by making \([\tilde{d}_j, \tilde{d}_I]\) a valid scalarization dependence matrix. Note that we use negative numbers for distances of antidepencences, positive numbers for those of true dependences. For the new matrix to be valid, we have

\[
\begin{align*}
(2n - 2m, 2) & \leq 0; \\
(-2n + m, -2) & \leq 0; \\
(-2m, 0) & \leq 0.
\end{align*}
\]

Solving these inequalities, we have the following constraint: \(n < m; m \leq 2n; m \geq 0.\) This makes \(n = 2; m = 3\) a valid integer solution. Thus, we use the formula \(j = 2I + 3J.\) The transformed code is shown in Figure 2.8.

```plaintext
DO I = 1, N
   DO j = 2*I+3, 2*I+3+M, 3
   ENDDO
ENDDO
```

Figure 2.8: Loop skewing example: \(j = 2I + 3J\)

Moving the \(j\) loop to the outermost position produces the code and the updated
dependence matrix shown in Figure 2.9. This is the final result we want to achieve.

```
DO j = 5, 2*N+3*M, 3
    DO I = max(I, (j-3*M)/2), min(3, (j-3)/2)
    ENDDO
ENDDO
```

Figure 2.9: Loop skewing example: move j outside and updated dependence matrix

### 2.4.2 Profitability and Applicability of Loop Skewing

The principle benefit of loop skewing is that it can avoid the use of a temporary array or at least reduce the dimensions of any required temporary array. In our example, if loop alignment is applied first, a temporary array of size $3 \times M$ or $2 \times N$ will have to be allocated, while loop skewing eliminates this space requirement. The penalty for loop skewing is variable loop bounds and skewed data access pattern. Therefore, loop skewing should probably be avoided in the innermost loop where stride-one access is desirable. In our examples, we used only two loops to demonstrate the mechanism of loop skewing in scalarization. In a real implementation, however, we should only consider outer loops as skewing candidates.

To determine whether loop skewing is applicable, we need to search all the combinations of two or more loops in the remaining loop nest in the worst case. The desired goal is to obtain a valid scalarization dependence matrix by solving a system of inequalities and finding an integer solution, in which case no temporary storage will be allocated. If such a solution does not exist, we try to make the first several columns of the matrix valid—a partial solution that still eliminates the temporary storage need
for the loops corresponding to the first several columns. The remaining loops can be further transformed using techniques such as loop reversal and loop alignment, as determined when necessary by the scalarization algorithm. An algorithm for searching loop skewing formula is given in Figure 2.10.

\textbf{procedure} loopskewing-formula:search\((L, D)\)
\hspace{1cm} // \(L = l_1, l_2, \ldots, l_k\), indices of the remaining loops
\hspace{1cm} // \(D = [\tilde{d}_1, \tilde{d}_2, \ldots, \tilde{d}_k]\), the corresponding scalarization dependence matrix,
\hspace{1cm} \(\tilde{d}_i\) is the column in \(D\) corresponding to \(l_i\)
\hspace{1cm} let \(\tilde{d} = c_1 \ast \tilde{d}_1 + c_2 \ast \tilde{d}_2 + \ldots + c_k \ast \tilde{d}_k\) where \(\{c_1, c_2, \ldots, c_k\}\) are variables
\hspace{1cm} for \(i = k\) to 1 do
\hspace{1.5cm} choose \(i\) loops \(l_{t_1}, l_{t_2}, \ldots, l_{t_i}\) from \(L\)
\hspace{1.5cm} for each choice of \(i\) loops do
\hspace{2.0cm} for \(j = 1\) to \(i\) do
\hspace{2.5cm} solve for integer solution \(\{c_1, c_2, \ldots, c_k\}\) such that
\hspace{2.5cm} \([\tilde{d}, \tilde{d}_{t_1}, \ldots, \tilde{d}_{t_{j-1}}, \tilde{d}_{t_{j+1}}, \ldots, \tilde{d}_{t_i}]\) is a valid scalarization matrix
\hspace{2.5cm} if an integer solution exists
\hspace{3.0cm} construct a new loop with index
\hspace{3.0cm} \(l_{t_j} = c_1 \ast l_1 + c_2 \ast l_2 + \ldots + c_k \ast l_k\) and corresponding bounds
\hspace{3.0cm} replace in loop body \(l_{t_j}\) with
\hspace{3.0cm} \((l_{t_j} - c_1 \ast l_1 - \ldots - c_{t_{j-1}} \ast l_{t_{j-1}} - c_{t_{j+1}} \ast l_{t_{j+1}} - \ldots - c_k \ast l_k) / c_{t_j}\)
\hspace{2.5cm} return \{\(l_{t_j}, t_j, l_1, t_2, \ldots, t_i\}\}
\hspace{1.5cm} return no-solution

Figure 2.10: Loop skewing formula search algorithm

Clearly, the size of search space is exponential in the number of loops in the loop nest. Furthermore, complicated formula will make the new loop have a non-unit step size. Fortunately, the number of array dimensions is typically small in practice, and we only consider as skewing candidates those loops that carry dependences. In any case, skewing requires a more expensive analysis than the other scalarization techniques discussed above.
2.4.3 Summary of Scalarization

As we pointed out in Section 2.2, when a sequence of scalarization transformations transforms a naively scalarized loop nest into semantically correct code, the original scalarization dependence matrix is changed into a valid one. Our scalarization turns this observation around by using the scalarization dependence matrix to drive the transformations. We summarize the effect of these transformations on the scalarization dependence matrix in Table 2.1, and present the overall scalarization algorithm in Figure 2.11.

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Effect on scalarization dependence matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop reversal</td>
<td>reverse (multiply by -1) the corresponding column</td>
</tr>
<tr>
<td>loop interchange</td>
<td>exchange corresponding columns</td>
</tr>
<tr>
<td>input prefetching</td>
<td>change current entry in the column to ( =_0 )</td>
</tr>
<tr>
<td>loop alignment</td>
<td>subtract align distance from the column</td>
</tr>
<tr>
<td>lazy loop alignment</td>
<td>subtract (align distance + 1) from the column</td>
</tr>
<tr>
<td>reduced lazy loop alignment</td>
<td>replace a column with a linear combination of itself and other columns</td>
</tr>
<tr>
<td>loop skewing</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Summary of scalarization transformations

2.5 Application: Stencil Computation

Stencil computations form a class of calculations that can benefit from loop alignment and loop skewing. It is natural and concise to write stencil computations in array syntax, so scalarization becomes an important tool for mapping them to the target machine.

In stencil computation, a problem domain is usually partitioned into grid points. The computation is iterated over these points. During each iteration, the value of
procedure scalarize(S)
// S: array assignment statement to be scalarized
naively scalarize S into a loop nest (LN)
compute the scalarization dependence matrix (SDM) based on LN
initialize resulting loop nest (RLN) to empty
while there are loops remaining in LN, from outermost to innermost loop
// case 1): loop interchange
if searching for a loop L by examining SDM such that L carries
no dependence, or only antidependences,
or only true dependences succeeds
if L carries only true dependences
apply loop reversal to L, multiply -1 to the column in SDM
remove L from LN and add L into RLN at innermost position
remove L (column) and dependences (rows) carried by L from SDM
continue
// case 2): loop skewing
\{L'_t, t_j, i, t_1, t_2, ..., t_k\} = loopskewing-formulasearch(LN, SDM)
if searching for L'_t succeeds
remove L'_t from LN and add L'_t into RLN at innermost position
remove L'_t (column) and dependences (rows) carried by L'_t from SDM
continue
// case 3): loop alignment
if searching for an alignable loop L by examining SDM succeeds
if loop alignment is applied for the first time
allocate temporary storage
apply loop alignment on L, add align distance to the column in SDM
remove L from LN and add L into RLN at innermost position
remove L (column) and dependences (rows) carried by L from SDM
continue
// case 4): failsafe method
if no loop alignment has been applied
allocate temporary storage
split the loop body into two loop nests, one for computation and
store into temporary, the other for store back to original array
break
end while
add remaining loop body in LN into RLN at innermost position
return RLN

Figure 2.11: Scalarization algorithm
a grid point is updated according to the old values of its neighboring points. This computation pattern will create both true dependences and antidepencences carried at different loop levels if the array syntax is to be scalarized naively. Therefore, temporary storage is needed to maintain the correct semantics. We will show that loop alignment and loop skewing can help reduce the temporary storage requirement in stencil computation. In fact, we will prove that scalarization with loop alignment and loop skewing can achieve asymptotically optimal (minimal) temporary storage requirement for a stencil calculation.

In what follows, we will establish lower bounds for the temporary storage requirements of stencil computations in Section 2.5.1 and Section 2.5.2. Based on this, we will prove that scalarization with loop alignment and loop skewing can achieve asymptotically optimal scalarizations for stencils in Section 2.5.3, assuming that dependence distances are known at compile time. We choose to separate two dimensional (2D) and multi-dimensional stencils because the proof for the multi-dimensional case requires the assistance of heavy use of mathematical notations, while in 2D case we can actually prove the theorem by enumerating all possible cases, thus easier to understand.

2.5.1 Minimal Temporary Storage Requirements for Two Dimensional Stencil Computations

Without loss of generality, we begin with a simple two-dimensional (2D) stencil in which the distance between a center point and a neighboring point is less than two. The theory applies to stencils with larger distances. First we define 2D stencil and its categories and computation.
Definition 2.5.1 A 2D stencil consists of a center point and one or more value-in neighbors whose old values are needed when computing the new value of the center point. A value-out graph will show the value-out neighbors to whom the center point will provide its old value. A 2D stencil with all nine points is called a full stencil, while a stencil with fewer than nine points is called a partial stencil. For a partial stencil, if there exists a straight line passing through the center point such that all stencil points lie on one side of the line (including on the line), then we say that the stencil is (1) a divisible stencil, (2) and the line is called a dividing line. (3) The side of a divisible stencil with all the points is called the value-in side, and (4) the side with no points is the value-out side. All other types of stencils (partial stencils with no dividing line, a full stencil) are called indivisible stencils. A minimal indivisible stencil is an indivisible stencil that will become divisible if any one value-in neighbor is removed. Notice that the dividing line for a divisible stencil may not be unique, but it doesn’t affect our theorem and its proof below. Figure 2.12 shows a full stencil (indivisible), a divisible stencil and its value-out graph.

![Stencil Diagram](image)

(a) full stencil  (b) partial & divisible stencil  (c) value-out graph of (b)

Figure 2.12: Stencil diagram

Definition 2.5.2 A 2D stencil computation written in array syntax has the following form:

\[ A(2:N+1, 2:N+1) = f(c0*A(1:N, 1:N), c1*A(1:N, 2:N+1), c2*A(1:N, 3:N+2), c3*A(2:N+1, 1:N), c4*A(2:N+1, 2:N+1), c5*A(2:N+1, 3:N+2), c6*A(3:N+2, 1:N), c7*A(3:N+2, 2:N+1), c8*A(3:N+2, 3:N+2)) \]
Before presenting the main theorem, we also need to introduce a bisection width theorem, proved in Leighton's text [37] and prove two lemmas that will be used in the proof.

**Definition 2.5.3** The *bisection width* of a graph is the minimum number of edges that have to be removed in order to disconnect the graph into two halves with identical (within one) numbers of points.

**Theorem 2.5.1** Every bisection of an $\overline{N \times N \times \cdots \times N}$ array contains at least $N^{r-1}$ edges.

**Proof** The proof appears in Leighton [37]. Note that an array in this theorem refers to a grid of points where edges are only parallel to the dimensions; *i.e.*, there are no diagonal edges.

**Lemma 2.5.2** The minimal temporary storage of computing a stencil, $S_1$, is always larger than or equal to that of computing any sub-stencil, $S_2$, which $S_1$ contains.

**Proof** We simply take a traversal order for computing $S_1$ and use it to compute $S_2$. Since the value-out neighbors of a point in $S_2$ are only a subset of those of the point in $S_1$, the temporary storage hold for the point in computing $S_2$ can always be released no later than it does in computing $S_1$. Therefore, computing $S_2$ requires no greater temporary storage than computing $S_1$.

**Lemma 2.5.3** The minimal temporary storage of computing a stencil $S$ in a 2D domain, $D_1$, is always larger than or equal to that of computing $S$ in a sub-domain $D_2$ of $D_1$. 

Proof  We simply take a traversal order for computing $S$ in $D_1$ and use it in $D_2$. Since the points keep the same relative ordering in $D_2$ as they do in $D_1$, computing $S$ in $D_2$ requires no greater temporary storage than doing so in $D_1$. $\blacksquare$

Now we are ready to prove the main theorem for 2D stencil computation.

**Theorem 2.5.4**  In 2D stencil computation, the minimal temporary storage requirement is $O(1)$ for a divisible stencil, and $O(N)$ for an indivisible stencil.

**Proof**  Case 1: Divisible stencil. As shown in Figure 2.13, there exists a dividing line that separates the stencil into value-in side and value-out side. For the center point, all value-in neighbors are on the value-in side (including the dividing line), all value-out neighbors are on the value-out side and the dividing line.

(\text{(a)}) (\text{(b)}) (\text{(c) dual stencil of (b)})

**Figure 2.13:** $O(1)$ case examples: divisible stencils

To achieve the $O(1)$ temporary storage requirement, an algorithm can be constructed as shown in Figure 2.14. Following this algorithm, when a point becomes the center point (computation reaches it), its dividing line coincides with the sweeping line, and its old value is only needed by its value-out neighbors that are either on the value-out side or on the dividing line. Since the computation for all points on the value-out side has already been performed, the old value of this point is only needed by its neighbors on the dividing line. Because the algorithm traverses the dividing line along one direction, there can be at most one value-out neighbor on the dividing line that has not been computed. That value-out neighbor is exactly the
next point that the algorithm will visit. After that value-out neighbor is computed, the old value of this point is no longer needed and its new value can be safely written back. Therefore, as shown in the algorithm, we only need two temporary variables to hold the new values.

\textbf{procedure} \textit{compute-divisible}

choose a dividing line for the divisible stencil

In parallel to the dividing line, sweep the 2D domain from the value-out side towards the value-in side, \textbf{for} each sweeping line

\textbf{for} each point along the sweeping line

perform the computation on the point (center point)

save the new value in a temporary variable ($t2$)

write the new value of previous point in temporary variable ($t1$) back

$t1 = t2$ (shift)

write the new value of last point in $t1$ back

Figure 2.14: Algorithm for divisible stencil: $O(1)$ temporary storage

The algorithm can always find a place to start because the domain is a bounded 2D space. Depending on the slope of the chosen dividing line, it either starts from one side of the boundary (loop alignment) or one corner (loop skewing plus loop alignment).

\textit{Case 2: Indivisible stencil.} As shown in Figure 2.15, no dividing line exists for indivisible stencils. Actually, (a), (b), (c), (d) and (e) show all five possible minimal indivisible stencils. (f) is isomorphic to (d) through rotation. First, we prove the existence of an algorithm that requires only $O(N)$ temporary storage for indivisible stencils, by presenting one in Figure 2.16. Similar to the $O(1)$ algorithm above for divisible stencil, this algorithm adopts a canonical sweeping line scheme. Instead of temporarily holding one new value, this algorithm holds all new values for the points on the current sweeping line. These new values can be safely written back after the
next line of points are visited since we are considering stencils in which the distance between a center point and its neighboring points is less than two.

![Stencils Diagram](image)

Figure 2.15: $O(n)$ case examples: indivisible stencils

**procedure** `compute-indivisible`

In parallel to one boundary, sweep the 2D domain from one side towards the other, **for** each sweeping line

- perform the computation for the points on the line
- save the new values in a temporary array ($t2(1:N)$)
- write the new value of previous line in temporary array ($t1(1:N)$) back
- $t1(1:N) = t2(1:N)$ (shift)
- write the new value of last line in $t1(1:N)$ back

Figure 2.16: Algorithm for indivisible stencil: $O(N)$ temporary storage

Next we prove that $O(N)$ temporary storage is minimal for indivisible stencils.

**Definition 2.5.4** At any snapshot of computation, there is a *visited frontier* that separates $N^2$ points into the visited part ($V$) and the unvisited part ($U$). For a visited point in $V$, it is:

- **on the visited frontier** if it has a neighbor in $U$;
- **external** if it has a value-out neighbor in $U$;
- **internal** if it is not external;
- **convex** if it has a horizontal or vertical neighbor in $U$; and
- concave if it is not convex and it has a diagonal neighbor in $U$ (Figure 2.17).

![Diagram](image)

Figure 2.17: $O(n)$ proof: visited frontier

From Definition 2.5.4, we can see that all external points in $V$ are on the visited frontier. For a full stencil, all points on the visited frontier are external. To prove that $O(N)$ temporary storage is minimal, we only need to show that the minimum number of external points is $O(N)$ at a certain snapshot for all visiting orders. More specifically, we will determine this lower bound at the bisection snapshot, i.e., when $V$ and $U$ have equal (within one) number of points. By Lemma 2.5.2, we only need to prove the theorem for all five types of minimal indivisible stencils as shown in Figure 2.15.

**Stencil (a):** This is the same stencil in Theorem 2.5.1 when $r = 2$. At any bisection snapshot, at least $N$ edges must be cut. If a point in $V$ has a edge connecting into $U$, then the point is an external point. Since the degree of each point is at most 4, there must be at least $N/4$ external points in $V$. Therefore, the minimal storage is at least $O(N)$.

**Stencil (b):** The 2D domain is separated into 2 sub-domains, each sub-domain is a connected group of roughly $N^2/2$ points. For any point in a sub-domain, all its value-in and value-out neighbors stay in the same sub-domain. For each sub-domain, we can rotate it by 45 degree, and carve out a square sub-sub-domain of size $N/2 \times N/2$ in the center as shown in Figure 2.18. In this sub-sub-domain, stencil topology is the
same as Stencil (a). Therefore, the minimal temporary storage in the sub-sub-domain is $O(N/2)$. By Lemma 2.5.3, the minimal temporary storage in the sub-domain is no less than $O(N/2)$. The two sub-domains have the same stencil topology, roughly equal size, therefore, the minimal temporary storage for computing Stencil (b) in the 2D domain is $O(N)$.

Figure 2.18: Stencil (b) case

Stencil (c), (d), (e): We will count the number of points on the visited frontier in $V$.

1) For any convex point on the visited frontier, it may be internal or external. However, the number of internal convex points is at most a constant multiple of the number of external points, because any internal convex point requires the existence of at least one external point nearby, and this external point can be shared by at most a constant number of other internal convex points. This can be proved by analyzing all possible cases as shown in Figure 2.19.

Figure 2.19: Stencil (c), (d), (e) cases, value-out graph
Figure 2.19 shows that A is an internal convex point. For two possible cases of stencil (c) shown in (c-1, c-2), if F is a unvisited point, then B and D must be external. However, if l1 is a boundary, then only B is external since D does not exist. If l2 is a boundary, then only D is external. Similarly, C or E must be external if G is a unvisited point. Therefore, for every internal convex point, there must be at least one external point among its neighbors. It is straightforward analog to prove this observation for 2 cases of stencil (d) shown in (d-2, d-3) and 2 cases of stencil (e) shown in (e-1, e-2).

One case remains for stencil (d) as shown in (d-1) where E is unvisited. In this case, if B is not external, then F must be already visited, and consequently F is external because it has a unvisited value-out neighbor: E. Similarly, if C is not external, then G must be external. Now we have proved that an internal convex point in all cases of stencil (c), (d) and (e) has at least one external point among its neighbors. Since one external point can be shared by at most 8 surrounding internal convex points, we can obtain the following inequalities for the number of convex points on the visited frontier:

\[
\#(\text{convex on visited frontier}) = \#(\text{external convex}) + \#(\text{internal convex}) \\
\leq \#(\text{external}) + \#(\text{internal convex}) \\
\leq \#(\text{external}) + 8 \times \#(\text{external}) \\
= 9 \times \#(\text{external})
\]

2) For any concave point on the visited frontier in V, it needs at least two neighboring convex points in V. Each one of these two convex points can be shared by at most one another concave point (Figure 2.17). Therefore, the maximum number of
concave points on the visited frontier cannot exceed that of convex points, which is $9 \times \#(external)$.

Based on 1) and 2), the total number of points on the visited frontier is at most $18 \times \#(external)$. For a full stencil, the number of points on the visited frontier is the same as that of external points. Since a full stencil contains stencil (a), by Lemma 2.5.2 and proof of Stencil (a), the number of points on the visited frontier is at least $N/4$ at the bisection snapshot. Thus for stencil (c), (d) and (e), the number of external points is at least $N/(18 \times 4)$, which is $O(N)$. □

**Definition 2.5.5** A *dual stencil* of a 2D stencil has the same center point and its value-in neighbors are exactly the value-out neighbors of the original 2D stencil. Examples are shown in Figure 2.13 (c) and Figure 2.15 (f).

**Corollary 2.5.5** The dual stencil of a divisible (indivisible) stencil is also a divisible (indivisible, respectively) stencil. The minimal temporary storage requirement for a stencil is the same as that for its dual stencil.

### 2.5.2 Minimal Temporary Storage Requirements for Multi-Dimensional Stencil Computations

The 2D theorem can be extended into higher dimensions. We write $S^M$ for a $M$ dimensional stencil in the following theorems. First, we look at the case that $S^M$ is divisible.

**Theorem 2.5.6** The minimal temporary storage requirement for computing a divisible stencil $S^M$ in $N^M$ space is equal to that for computing $S^{M-1}$ in $N^{M-1}$ space,
where $S^{M-1}$ is a sub-stencil of $S^M$ that resides on a $M - 1$ dimensional "straight" (linear) dividing structure for $S^M$.

**Proof** The dividing structure divides the $N^M$ space into two parts, for a point on the dividing structure, all its value-in neighbors reside in only one part. Therefore, similar to the strategy shown in Figure 2.14, we can sweep the dividing structure across the $N^M$ domain from the value-out side towards the value-in side, and perform the computation on each sweeping structure. By this sweeping order, the temporary storage for a point can be released as soon as its value-out neighbors on the same sweeping structure are all visited. Hence, the minimal temporary storage requirement for computing $S^M$ in $N^M$ is less than or equal to that for computing $S^{M-1}$ in $N^{M-1}$.

On the other hand, $S^{M-1}$ is a sub-stencil of $S^M$ and $N^{M-1}$ is a sub-domain of $N^M$. By Lemma 2.5.2 and Lemma 2.5.3, the minimal temporary storage requirement for computing $S^M$ in $N^M$ is larger than or equal to that for computing $S^{M-1}$ in $N^M$, which in turn is larger than or equal to that for computing $S^{M-1}$ in $N^{M-1}$. □

**Corollary 2.5.7** If there are two different dividing structures for $S^M$ in $N^M$ space, let $S_1^{M-1}$ and $S_2^{M-1}$ be the two sub-stencils of $S^M$ that reside on the two dividing structures, respectively, then 1) the minimal temporary storage for computing $S_1^{M-1}$ in $N^{M-1}$ space is the same as that for computing $S_2^{M-1}$ in $N^{M-1}$; 2) moreover, they all equal to that for computing $S_1^{M-1} \cap S_2^{M-1}$ in $N^{M-2}$.

**Proof** Part 1 of the conclusion is a direct result of Theorem 2.5.6. For part 2, let the two dividing structures be $X_1$ and $X_2$ and their intersection be $X_1 \cap X_2$ (they must intersect since they all pass through the center point, and the intersection is a $N^{M-2}$ structure). $X_1$ is a dividing structure for $S^M$, so it is for a sub-stencil $S_2^{M-1}$
in $N^M$. Since $S_2^{M-1}$ resides on $X_2$ only, $X_1 \cap X_2$ is a dividing structure for $S_2^{M-1}$ in $X_2$. Similarly, $X_1 \cap X_2$ is also a dividing structure for $S_2^{M-1}$ in $X_1$. Moreover, the sub-stencil of $S_1^{M-1}$ on $X_1 \cap X_2$ and the sub-stencil of $S_2^{M-1}$ on $X_1 \cap X_2$ are exactly the same: $S_1^{M-1} \cap S_2^{M-1}$. ■

For an indivisible stencil $S^M$ in $N^M$ space, it is easy to see that a sweeping algorithm similar to the one shown in Figure 2.16 exists, and it requires $O(N^{M-1})$ temporary storage. However, it is quite hard to extend our 2D case by case strategy to prove that $O(N^{M-1})$ is minimal. Fortunately, a closely related problem, minimal edge boundary for a star stencil (a center point only has two neighbors on each dimension, one on each side, and no diagonal neighbors) in grids, has been studied under a mathematical term -- isoperimetric inequalities ([52], [9] and [8]). For a star stencil, in Bollobás and Leader [8], the boundary of a set of points is always larger than or equal to that of the same number of points in a compact region — all points are pushed (compressed) towards one corner along dimensions and a set of vectors; while Corollary 4 in another paper by Bollobás and Leader [9] showed that the minimal edge boundary between a set and its complement in $N^M$ is $N^{M-1}$ when the number of points in the set is between a quarter and three quarters of $N^M$. However, their approach cannot be directly applied to our case since our general stencil topology contains diagonal edges. But by changing the definitions and formulas, we can use a similar compression technique to prove our theory about the minimal temporary requirement for indivisible stencils.

First, let’s introduce notations to be used in the proof. These are very similar to the ones used in Bollobás and Leader [8], with slight twist to accommodate our situation. Given a stencil $S^M$ and domain $N^M$, let $E$ be the value-out relations (edges) among the points, for a visited set $A$, we write $\partial A$ for one-edge expansion
(boundary) of $A$ by definition: $\partial A = \{x : x \in A \mid \exists y \in A \text{ and } yx \in E\}$. Let $e_1, \ldots, e_M$ be the standard basis of $Z^M$, then any value-out edge of $S^M$ is a vector in the form of: $v = c_1 e_1 + \cdots + c_M e_M$, where $c_i \in \{-1, 0, 1\}$ and $\exists c_i \neq 0$. For each value-out edge $v$, we define the lower boundary of $N^M$ along $v$ to be $N^v = \{x \in N^M : x + v \not\in N^M\}$. For each value-out edge $v$ and $x \in N^v$, we define the $v$-section of $A$ at $x$ to be $A_v(x) = \{i \geq 0 : x - iv \in A\}$, the upper boundary of $v$-section at $x$ to be $u_v(x) : x - u_v(x)v \in N^M$ and $x - (u_v(x) + 1)v \not\in N^M$, and we define $C_v(A)$, the $v$-compression of $A$ by giving its $v$-sections ($x \in N^v$):

$$C_v(A)_v(x) = \begin{cases} 
\{0, 1, \ldots, |A_v(x)| - 2, u_v(x)\}, & \text{if } u_v(x) \in A_v(x) \\
\{0, 1, \ldots, |A_v(x)| - 1\}, & \text{otherwise}
\end{cases}$$

Intuitively, $C_v(A)$ is the result of compressing all points of $A$ along the direction $v$ towards the lower boundary, while leaving the points on the upper boundary unmoved if there are any. Obviously, the number of points after compression remains the same; i.e., $|C_v(A)| = |A|$. We say $A$ is $v$-compressed if $C_v(A) = A$.

**Lemma 2.5.8** For a minimal indivisible stencil $S^M$, let $v$ be a value-out edge, a point on a $v$-section has at most one value-out edge to a neighboring $v$-section.

**Proof** Suppose not. Let the point be $P_1$, its value-out neighbor on the $v$-section be $P_2$, assume $P_1$ has two value-out neighbors $P_3$ and $P_4$ on another $v$-section, as shown in Figure 2.20. In all cases, if any dividing structure passing through $P_1$ separates $P_2$ and $P_4$ or $P_4$ and $P_3$, it will also separate $P_2$ and $P_3$. Therefore, $P_4$ is redundant and removing $P_4$ does not make $S^M$ divisible. This contradicts to the definition of a minimal indivisible stencil. ■
Theorem 2.5.9  The minimal temporary storage requirement for computing an indivisible stencil $S^M$ in $N^M$ space is $O(N^{M-1})$.

Proof  First of all, we consider only $S^M$ such that $N^M$ points are strongly connected through undirectional value-out edges. Otherwise, we can always partition $N^M$ into several strongly connected subsets and reduce to a computation on a subset of same asymptotic size, by similar techniques for proving 2D case shown in Figure 2.15 (b).

By Lemma 2.5.2, we only need to prove for minimal indivisible stencils. Let $S^M$ be minimal indivisible.

Given a visited set $A$ in $N^M$, we are going to show that for each value-out edge $v$, $|\partial A| \geq |\partial C_v(A)| - w_v$, where $w_v$ is a number that will be defined later. To do that, we prove that for each $x \in N^k$, $|(\partial C_v(A))_v(x)| \leq |(\partial A)_v(x)| + 3$. The one-edge expansion of $A$ at $x$ along vector $v$ comes from two part: the one-edge expansion of $A_v(x)$ itself, and the one-edge expansion from neighboring $v$-sections ($y \in N^k$):

$$(\partial A)_v(x) = (\partial (A_v(x)))_v(x) \cup \bigcup_{y \neq x} (\partial (A_v(y)))_v(x)$$
A similar relation holds for $C_v(A)$:

$$(\partial C_v(A))_v(x) = (\partial (C_v(A)_v(x)))_v(x) \cup \bigcup_{y \neq x} (\partial (C_v(A)_v(y)))_v(x)$$

Part 1) For the points expanded by $A_v(x)$ itself, it is easy to verify that $|\partial (C_v(A)_v(x)))_v(x)| = |\partial (A_v(x)))_v(x)| + 1$ happens only when all points in $A_v(x)$ are compacted towards the upper boundary and $-v$ is also a value-out edge, as shown in Figure 2.21 (a). However, in this case, current $v$-section always contains an external point. For other cases, $|\partial (C_v(A)_v(x)))_v(x)| \leq |\partial (A_v(x)))_v(x)|$.

Part 2) For the points expanded by neighboring $v$-sections, by Lemma 2.5.8, each point on a neighboring $v$-section at point $y$ has at most one value-out neighbor on this $v$-section at point $x$. If $(\partial C_v(A_v(y)))_v(x)$ is a continuous set starting from 0, i.e., $(\partial (C_v(A_v(y)))_v(x) = \{0, 1, \cdots, |\partial (C_v(A_v(y)))_v(x)| - 1\}$ for all $y$, then $|\bigcup_{y \neq x} (\partial (C_v(A)_v(y)))_v(x)| \leq |\bigcup_{y \neq x} (\partial (A_v(y)))_v(x)|$.

However, $(\partial (C_v(A_v(y))))_v(x)$ may not be a continuous set starting from 0 because of either a neighboring backward value-out edge (let it be $v'$, then the dot product $v \cdot v' \leq 0$) (Figure 2.21 (b)) or a neighboring point at the upper boundary (Figure 2.21 (c)) or both (Figure 2.21 (d) (e)). In these cases, the difference $|\partial (C_v(A)_v(x)| - |\partial A_v(x)|$ is 1 or 2. The largest difference is 3 when the effect of a backward edge is combined with difference from Part 1 (Figure 2.21 (f)). Nevertheless, close examination of all these cases reveal that there must be an external point either in the current $v$-section or a neighboring $v$-section before compression. For all other cases, $|\partial (C_v(A)_v(x)| \leq |\partial A_v(x)|$. Notice that the analysis above is also true if $v$ is skewed, i.e., $v \notin \{e_1, \cdots, e_M\}$, an example is given in Figure 2.21 (g).
Based on Part 1) and Part 2), we define

\[
    w_v(x) = \begin{cases} 
        |(\partial C_v(A))_v(x)| - |(\partial A)_v(x)|, & \text{if } |(\partial C_v(A))_v(x)| > |(\partial A)_v(x)| \\
        0 & \text{otherwise}
    \end{cases}
\]

and \(w_v = \sum_{x \in N^v} w_v(x)\), now we have \(w_v(x) \leq 3\), and \(|\partial A| \geq |\partial C_v(A)| - w_v\). Notice that \(|C_v(A)| = |A|\) and \(A \subseteq \partial A\), we have \(|\partial A| - |A| \geq |\partial C_v(A)| - |C_v(A)| - w_v\), and hence \(|\partial A - A| \geq |\partial C_v(A) - C_v(A)| - w_v\). Let the value-out edges (if both \(v\) and \(-v\) are value-out edges, pick only one of them) be \(v_1, \cdots, v_k\), we can compress \(A\) along these edges and obtain a set \(A' = C_{v_k}(\cdots(C_{v_2}(C_{v_1}(A)))\cdots)\). Thus

\[
    |\partial A - A| \geq |\partial (C_{v_1}(A)) - C_{v_1}(A)| - w_{v_1}
\]

\[
    \cdots
\]

\[
    \geq |\partial C_{v_2}(\cdots(C_{v_1}(A))\cdots) - C_{v_2}(\cdots(C_{v_1}(A))\cdots)| - \sum_{i \leq j} w_{v_i}
\]
\[ \geq |\partial A' - A'| - \Sigma_{i \leq k} w_{v_i} \]

To prove \(O(N^{M-1})\) is minimal for \(S^M\), it is sufficient to show that for any visited set \(A\) containing \(N^M/2\) points, \(|\partial A - A|\) is at least \(O(N^{M-1})\).

1) If \(\Sigma_{i \leq k} w_{v_i}\) is at least \(O(N^{M-1})\), let \(j\) be the first \(w_{v_j}\) that is at least \(O(N^{M-1})\), then at least \(w_{v_j}/3\) \(v_j\)-sections have an external point either in itself or a neighboring \(v_j\)-section in \(C_{v_{j-1}}(\cdots (C_{v_1}(A))\cdots)\).

Thus \(|\partial C_{v_{j-1}}(\cdots (C_{v_1}(A))\cdots) - C_{v_{j-1}}(\cdots (C_{v_1}(A))\cdots)|\) is at least \(O(N^{M-1})\) and so is \(|\partial A - A|\).

2) Now suppose \(\Sigma_{i \leq k} w_{v_i}\) is less than \(O(N^{M-1})\). We only need to show that \(|\partial A' - A'|\) is at least \(O(N^{M-1})\). In Bollobás and Leader [9], for a star stencil, the minimum number of edges between a set \(B\) and \(\partial B - B\) is \(N^{M-1}\) when \(N^M/4 \leq |B| \leq 3N^M/4\). Hence for \(A'\), there are at least \(O(N^{M-1})\) points that has a unvisited neighbor along some dimension. Since \(S^M\) is indivisible, \(v_1, \cdots, v_k\) should cover all the basis \(e_1, \cdots, e_M\). Thus the points in \(A'\) have been compressed against one corner or one dimension plus some points on upper boundaries, at least half of the \(O(N^{M-1})\) points are external points for \(A'\) as shown in Figure 2.21 (h) since the other half could be internal.

### 2.5.3 Optimal Scalarization for Stencil Calculations

We now consider how loop alignment and loop skewing can be applied to achieve the optimal temporary storage in scalarizing stencil computation. We begin with two examples that illustrate the issues.
The stencil in the following code

\[ A(2:N+1, 2:N+1) = A(1:N, 1:N) + A(3:N+2, 1:N) + A(1:N, 3:N+2) \]

has a topology shown in Figure 2.13 (a), and thus is a divisible stencil. The naively scalarized version and the scalarization dependence matrix is shown in Figure 2.22 and Figure 2.23 (a).

```
DO J = 1, N
  DO I = 1, N
    A(I+1, J+1) = A(I, J) + A(I+2, J) + A(I, J+2)
  ENDDO
ENDDO
```

Figure 2.22: O(1) case: naively scalarized code

By loop skewing alone, there is no integer solution for \( m, n \) such that \([n \ast \vec{d}_I + m \ast \vec{d}_J, \vec{d}_I]\) or \([n \ast \vec{d}_I + m \ast \vec{d}_J, \vec{d}_J]\) is a valid scalarization dependence matrix. However, we can combine loop skewing and loop alignment to achieve this goal. First, we apply loop skewing to obtain a valid \( n \ast \vec{d}_I + m \ast \vec{d}_J \) column in the matrix. Second, we use loop alignment to make the \( \vec{d}_I \) column valid. We choose formula \( j = -I - J \) for loop skewing and distance 1 for loop alignment on \( I \). The intermediate and final scalarization matrices and the final code are shown in Figure 2.23 and Figure 2.24, respectively. The algorithm uses only \( O(1) \) temporary storage, which is the optimal, as we have previously shown.

\[
\begin{bmatrix}
  J & I \\
  <_1 & <_1 \\
  <_1 & >_{-1} \\
  >_{-1} & <_1
\end{bmatrix}
\begin{bmatrix}
  j & I \\
  >_{-2} & <_1 \\
  =_0 & >_{-1} \\
  =_0 & <_1
\end{bmatrix}
\begin{bmatrix}
  j & I \\
  >_{-2} & \equiv_0 \\
  =_0 & >_{-2} \\
  =_0 & \equiv_0
\end{bmatrix}
\]

(a) original (naive)  (b) skewing: \( j = -I - J \)  (c) alignment: distance 1 on \( I \)

Figure 2.23: O(1) case: intermediate and final scalarization dependence matrices

Our second example is a full 2D stencil, which is therefore indivisible.
DO j = -2N, -2
Tl = max(1, -j-N)
T(1) = A(Tl, -j-Tl) + A(Tl+2, -j-Tl) + A(Tl, -j-Tl+2)
DO l = max(1, -j-N), min(N, -j-1) - 1
T(2) = A(l+1, -j-l-1) + A(l+3, -j-l-1) + A(l+1, -j-l+1)
A(l+1, -j-l+1) = T(1)
T(1) = T(2)
ENDDO
Tl = min(N, -j-1)
A(Tl+1, -j-Tl+1) = T(1)
ENDDO

Figure 2.24: O(1) case: final scalarized code

\[
\]

Loop alignment determines that the alignment distance is 1 and therefore a temporary array of size 2*2 is allocated. As shown in Figure 2.25, the final code uses a temporary array of optimal size. If lazy loop alignment (reduced lazy loop alignment) is chosen, the algorithm only needs to align the outer loop with a temporary array of size 3 * N (2 * N), as shown in Figure 2.26 (Figure 2.27, respectively). Note that the lazy loop aligned code and reduced lazy aligned code have perfectly nested loops compared to the loop aligned code.

These examples lead us to ask whether it is always possible to achieve the asymptotically optimal storage requirements using our scalarization strategies (Figure 2.11). The following theorem resolves this question.

**Theorem 2.5.10** The scalarization algorithm shown in Figure 2.11 will produce asymptotically optimal scalarization for stencil computation.

**Proof** For a M dimensional stencil computation in a N^M space, there are M loops in naively scalarized loop nest. The algorithm allocates no temporary storage until a loop is chosen during the loop alignment phase. Before that, if there are K
ALLOCATE T(N, 2)
DO I = 1, N
  T(I, 1) = A(I, 1) + A(I, 2) + A(I, 3) + A(I+1, 1) + A(I+1, 2) + A(I+1, 3)
  = A(I+2,1) + A(I+2,2) + A(I+2,3)
ENDDO
DO J = 1, N-1
  T(1, 2) = A(1, J+1) + A(1, J+2) + A(1, J+3) + A(2, J+1) + A(2, J+2) + A(2, J+3)
  = A(3, J+1) + A(3, J+2) + A(3, J+3)
  DO I = 1, N-1
    T(I+1, 2) = A(I+1, J+1) + A(I+1, J+2) + A(I+1, J+3) + A(I+2, J+1) + A(I+2, J+2) + A(I+2, J+3)
    + A(I+3, J+1) + A(I+3, J+2) + A(I+3, J+3)
    A(I+1, J+1) = T(I, 1)
    T(I, 1) = T(I, 2)
  ENDDO
  A(N+1, J+1) = T(N, 1)
  T(N, 1) = T(N, 2)
ENDDO
DO I = 1, N
  A(I+1, N+1) = T(I, 1)
ENDDO

Figure 2.25: Full 2D stencil scalarized with loop alignment

loops get chosen by either the loop interchange phase or the loop skewing phase, then only \( M - K \) loops are remaining when the loop alignment phase is encountered. Hence, a temporary storage of \( O(N^{M-K-1}) \) is allocated. Now we prove that each choice of these \( K \) loops defines a dividing structure at different levels (from \( N^M \) to \( N^{M-K+1} \) in space).

For the loop chosen by loop interchange or created by loop skewing, 1) either it carries no dependence, thus at each iteration of this loop, all value-in neighbors are visited at the same iteration that the center point is visited; 2) or it carries only antidependences (or true dependences), thus at each iteration of this loop, all value-in neighbors are visited at the later (or earlier, respectively) iterations or at the same iteration that the center point is visited. Therefore, every single iteration of the chosen loop defines a dividing structure such that the remaining value-in neighbors exist on only one side of the structure. Since the algorithm always ensures that the emitted loop carries no true dependences (loop reversal is applied when necessary),
ALLOCATE T(N, 3)
DO I = 1, N
  T(I, 1) = A(I, 1) + A(I+1, 2) + A(I+1, 3) + A(I+1, 1) + A(1, 2) + A(I+1, 3)
  + A(I+2, 1) + A(1+2, 2) + A(I+2, 3)
  T(I, 2) = A(I, 2) + A(I, 3) + A(I, 4) + A(I+1, 2) + A(I+1, 3) + A(I+1, 4)
  + A(I+2, 2) + A(I+2, 3) + A(I+2, 4)
ENDDO
DO J = 1, N-2
  DO I = 1, N
    T(I, S) = A(I, J+2) + A(I, J+3) + A(I, J+4) + A(I+1, J+2) + A(I+1, J+3) + A(I+1, J+4)
    + A(I+2, J+2) + A(I+2, J+3) + A(I+2, J+4)
    A(I+1, J+1) = T(I, 1)
    T(I, 1) = T(I, 2)
    T(I, 2) = T(I, S)
  ENDDO
ENDDO
DO I = 1, N
  A(I+1, N) = T(I, 1)
  A(I+1, N+1) = T(I, 2)
ENDDO

Figure 2.26: Full 2D stencil scalarized with lazy loop alignment
the emitted loop iterates from the value-out side towards the value-in side.

Therefore, the algorithm exhaustively searches for a dividing structure at each
level. By Corollary 2.5.7, it does not matter which dividing structure the algorithm
takes when there are multiple choices, since they lead to the same minimal temporary
storage requirement. Naturally, our algorithm prefers an original loop found by the
loop interchange phase than a skewed loop created by the loop skewing phase due
to the associated cost. When a dividing structure cannot be found anymore, by
Theorem 2.5.9, the algorithm allocates a minimal temporary storage. ■

2.6 Experiments

We implemented loop alignment in the D System developed at Rice University. Pre-
viously, the scalarization code included loop reversal, loop interchange and the failsafe
method. We did not implement loop skewing because of the complexity of the asso-
ciated analysis. The tool we constructed is basically a source to source transformer.
ALLOCATE T(N, 2)
DO I = 1, N
  T(I, 1) = A(I, 1) + A(I, 2) + A(I, 3) + A(I+1, 1) + A(I+1, 2) + A(I+1, 3)
  T(I, 2) = A(I, 2) + A(I, 3) + A(I, 4) + A(I+1, 2) + A(I+1, 3) + A(I+1, 4)
    + A(I+2, 2) + A(I+2, 3) + A(I+2, 4)
ENDDO
DO J = 1, N-2
  DO I = 1, N
    A(I+1, J+1) = T(I, 1)
    T(I, 1) = T(I, 2)
    T(I, 2) = A(I, J+2) + A(I, J+3) + A(I, J+4) + A(I+1, J+2) + A(I+1, J+3) + A(I+1, J+4)
      + A(I+2, J+2) + A(I+2, J+3) + A(I+2, J+4)
  ENDDO
ENDDO
DO I = 1, N
  A(I+1, N) = T(I, 1)
  A(I+1, N+1) = T(I, 2)
ENDDO

Figure 2.27: Full 2D stencil scalarized with reduced lazy loop alignment

Given a Fortran 90 program written in array syntax, our tool will scalarize array syntax into loop nests, while keeping other parts of the program unchanged. The transformed program can then be passed to commercial compilers for code generation.

2.6.1 Loop Alignment

Our experiment uses a Jacobi iterative method for solving partial differential equations (PDEs) using a nine-point stencil. The major computation takes part in the following array assignment statement, which is iterated 400 times.

\[
grid(2:N+1, 2:N+1) = grid(1:N, 1:N) + grid(1:N, 2:N+1) + grid(1:N, 3:N+2)
+ grid(2:N+1, 1:N) + grid(2:N+1, 2:N+1) + grid(2:N+1, 3:N+2)
+ grid(3:N+2, 1:N) + grid(3:N+2, 2:N+1) + grid(3:N+2, 3:N+2))
/ 9.0
\]

To scalarize this statement, the failsafe method generates a full size \((N \times N)\) temporary array. The results of this computation are stored into the temporary array first, then copied back into grid. In a Compaq Fortran memory management tutorial,
the presenters actually allocated two copies of \texttt{grid} that were called \texttt{new} and \texttt{old}, at the end of each iteration, \texttt{new} and \texttt{old} are switched. By doing so, the copying operation in the failsafe method is eliminated. However, if loop alignment (or lazy loop alignment) is applied, we can reduce the temporary size to $2 \times N$ (or $3 \times N$, respectively). The loop alignment version and lazy loop alignment version of a full 2D stencil are given as examples in Figure 2.25 and Figure 2.26.

We tested six versions of the Jacobi code (nine-point stencil): the failsafe version, the Compaq version, the loop alignment version, the lazy loop alignment version, together with the loop alignment and lazy loop alignment versions that use loop unrolling to remove array copying. They are referred to as “failsafe”, “compaq”, “align”, “lalign”, “align+url” and “lalign+url” in the following performance figures, respectively. These six versions were compiled by SGI MIPSpro compiler (version 7.3.1m) with highest optimization level (O3) and tested on a 195MHz SGI O2 workstation. We measured running time, TLB misses, L1 and L2 cache misses on different grid sizes: $200 \times 200$, $400 \times 400$, $800 \times 800$ and $1600 \times 1600$. We want to show that by reducing the temporary array size, the memory hierarchy performance can be improved.

From Figure 2.28 (a), we can see that the loop alignment version achieves speedups of 1.66 over the Compaq version and 1.62 over the failsafe version, which is consistent with our expectations. However, to validate our hypothesis, we must determine whether the memory hierarchy contributed to these speedups. The number of TLB misses in all alignment versions are substantively smaller than those of the other two versions as shown in Figure 2.28 (b). In particular, the alignment versions incur only half as many TLB misses as the Compaq double grid switch version, and seven times fewer than that of the version using the full sized temporary array. In Figure 2.28
Figure 2.28: Performance data on SGI (normalized to failsafe version, nine-point stencil)

(c), the primary (L1) cache misses show a similar pattern until grid reaches the size of 1600x1600. However, when grid exceeds that size, secondary cache misses become the dominant factor. From Figure 2.28 (d), we can see that the secondary cache miss rates in the alignment versions are a factor of two less than those of the other two versions for large grid sizes.

Aside from the effect of reducing temporary memory size, we can make two more observations from Figure 2.28 (a): (1) Removing array copying operations with loop unrolling helps improve the performance. (2) Lazy loop alignment version with loop
unrolling actually performs better than loop alignment version with loop unrolling. This is also consistent with our speculation in Section 2.3.4 that the contributing factors to performance includes not only memory footprint size, but also the complexity of source code, which affects later phases of compilation. These two observations lead us to investigate array copying elimination with rotating array indices (with MOD operators), along with reduced lazy loop alignment.

2.6.2 Commercial Compilers

We also investigated four commercial compilers from SGI, Compaq (Alpha), SUN and Intel. We fed the commercial compilers eight different versions of nine-point stencil Jacobi code: original array syntax version, the failsafe version, versions of using loop unrolling and versions of using MOD operators for loop alignment, lazy loop alignment and reduced lazy loop alignment, respectively. They are denoted in Figure 2.29 as “orig”, “failsafe”, “align-url”, “align+MOD”, “lalign-url”, “lalign+MOD”, “reddie-lalign-url” and “reddiealign+MOD”, respectively. We measured the running time of each version on the following compiler and architecture combinations: SGI MIPSPRO 7.3.1m (optimization level O3) on a 195MHz SGI O2 workstation, DIGITAL Fortran V5.2 (O5) on a 500MHz Compaq server, SUNWspro 7.2 (O3) on a 500MHz SUN Blade workstation, and Intel Fortran compiler 7.0 (O3) on a 2.4GHz Pentium 4 workstation running Linux.

From Figure 2.29, we can clearly see that for SGI, Compaq and Intel compilers, loop alignment optimized code performs better than the original code, from which we can speculate that these three compilers did not implement the loop alignment strategy. We can reach the same speculation by the fact that on all machines the
original code performs almost identical to the failsafe version, which allocates a full size temporary array. For SUN, the loop aligned code performs even worse than the original code. We believe that this is due to the complexity of the loop aligned code, which is not perfectly nested and thus probably affects the instruction scheduling phase in the backend compiler. This becomes clear by comparing the performance and the actual code (Figure 2.25 and 2.27) of the loop aligned version and the reduced lazy loop aligned version. Nevertheless, for all four compilers, the lazy loop aligned versions achieved speedups in the range of 1.56 to 2.20 over the original code, and
they generally perform the same as or better than the loop aligned versions. The performance difference between code using loop unrolling and code using rotating array indices (MOD) is within sixteen percent.

In addition to the nine-point stencil, we also tested a five-point stencil Jacobi code as follows,

\[
\text{grid}(2:\text{N}+1, 2:\text{N}+1) = \left(\text{grid}(1:\text{N}, 2:\text{N}+1) + \text{grid}(2:\text{N}+1, 1:\text{N}) + \text{grid}(2:\text{N}+1, 3:\text{N}+2) + \text{grid}(3:\text{N}+2, 2:\text{N}+1)\right) / 4.0
\]

with same versions of code on the same four compiler and architecture combinations. From Figure 2.30, we can see that our loop alignment strategy achieved an even better result on the five-point stencil than it did on the nine-point stencil. The speedup of our best alignment version over the original version is in a range of 1.97 to 2.97.

### 2.6.3 Fine Tuning Loop Alignment

While the main purpose of our general loop alignment strategy is to reduce the temporary storage size generated during the scalarization, we observe that the generated code interferes with backend compiler to some extent, although we do not completely understand the details. For example, reduced lazy loop alignment seems to have the benefits of both loop alignment (same size of allocated temporary) and lazy loop alignment (clean code), however, its unrolled version doesn’t always perform better than unrolled versions of other two alignments. For the nine-point stencil Jacobi code, since lazy loop alignment unrolls three iterations to remove array copying, while loop alignment and reduced lazy loop alignment each unroll two iterations, we want to investigate the case of unrolling six iterations (the least common multiple). In Figure 2.31, suffix "+url6" indicates the code that unrolls six iterations. We can see that
Figure 2.30: Commercial compilers, running time (normalized to original version, five-point stencil)

on SGI, all three alignment versions run for about the same amount of time after unrolling six iterations, and they all improve compared to the corresponding versions with or without original unrolling. On the SUN, unrolling six iterations has no uniform effects. For grid size 1600 × 1600, reduced lazy loop alignment without loop unrolling actually performs the best. On Compaq and Intel, unrolling six iterations has no effect, the performance remains the same as the ones with original unrolling.

We took our experiments to a VLIW architecture available to us: 64-bit Intel Itanium 2. Similar to other architectures, we fed six versions of nine-point stencil
Figure 2.31: Fine tuning for backend compiler: loop unrolling (normalized to align version, nine-point stencil)

Jacobi code: “orig”, “failsafe”, “lalign-url”, “lalign+MOD”, “redlalign+url” and “redlalign+MOD” to Intel Fortran Compiler 7.0 (O3). To our surprise, even though our scalarization strategy reduced the number of cache misses on all levels of memory hierarchy dramatically, however, the best speedup over the original version was only about 1.18 as shown in Table 2.2. In fact, the number of floating point instructions retired actually increased for code produced by our scalarization algorithm. Note that L1 cache is for integers only on Itanium 2. Since Itanium 2 is a VLIW machine

<table>
<thead>
<tr>
<th>Program</th>
<th>Running time</th>
<th>L2 misses</th>
<th>L3 misses</th>
<th>FP instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>orig</td>
<td>2.63 secs</td>
<td>64927045</td>
<td>64603475</td>
<td>2113280688</td>
</tr>
<tr>
<td>failsafe</td>
<td>2.60 secs</td>
<td>50710360</td>
<td>59107879</td>
<td>2113280688</td>
</tr>
<tr>
<td>align-url</td>
<td>4.14 secs</td>
<td>16641360</td>
<td>16365183</td>
<td>2303523685</td>
</tr>
<tr>
<td>align+MOD</td>
<td>2.22 secs</td>
<td>16619680</td>
<td>16420002</td>
<td>2303523685</td>
</tr>
<tr>
<td>redlalign-url</td>
<td>2.53 secs</td>
<td>16637393</td>
<td>16385797</td>
<td>2303523685</td>
</tr>
<tr>
<td>redlalign+MOD</td>
<td>3.20 secs</td>
<td>16630431</td>
<td>16446257</td>
<td>2303523685</td>
</tr>
<tr>
<td>lalign+MOD+dist</td>
<td>1.44 secs</td>
<td>16600184</td>
<td>16378218</td>
<td>1538720485</td>
</tr>
<tr>
<td>redlalign+MOD+dist</td>
<td>1.43 secs</td>
<td>16618676</td>
<td>16422454</td>
<td>1538720485</td>
</tr>
</tbody>
</table>

Table 2.2: Hardware counter readings for grid size 800 × 800 on Itanium 2
on which software pipelining is a very important optimization, we suspected that our code was not well handled during software pipelining in the backend compiler. To investigate this issue, we distributed the inner loop around its statements (Figure 2.32), which is always safe because there is no dependence cycle in our scalarized code. The performance result is shown in Figure 2.33, where suffix "dist" indicates

```
ALLOCATE T(N, 2)
DO I = 1, N
  T(I, 1) = A(I, 1) + A(I, 2) + A(I, 3) + A(I+1, 1) + A(I+1, 2) + A(I+1, 3)
    + A(I+2, 1) + A(I+2, 2) + A(I+2, 3)
  T(I, 2) = A(I, 2) + A(I, 3) + A(I, 4) + A(I+1, 2) + A(I+1, 3) + A(I+1, 4)
    + A(I+2, 2) + A(I+2, 3) + A(I+2, 4)
ENDDO
DO J = 1, N-2
  DO I = 1, N  ! loop distribution
    A(I+1, J+1) = T(I, MOD(I+1,2)+1)
  ENDDO
  DO I = 1, N  ! loop distribution
    T(I, MOD(J+1,2)+1) = A(I, J+2) + A(I, J+3) + A(I, J+4) + A(I+1, J+2) + A(I+1, J+3)
      + A(I+1, J+4) + A(I+2, J+2) + A(I+2, J+3) + A(I+2, J+4)
  ENDDO
ENDDO
DO I = 1, N
  A(I+1, N) = T(I, MOD(N-1,2)+1)
ENDDO
```

Figure 2.32: Full 2D stencil scalarized with reduced lazy loop alignment, rotating array indices and loop distribution

the loop distributed version. Now both new versions achieved a speedup of 1.80 over the original version. Readings from the hardware counters indicate that the number of floating point instructions retired are reduced after loop distribution. However, without knowing the details inside the Intel compiler, we can only conclude that it compiles distributed loops better than fused loops in our case. Nevertheless, it does not affect the main conclusion: our scalarization strategy helps improve the memory hierarchy performance and consequently the running time. While running distributed versions on previous four architectures mentioned in Section 2.6.2, we did not observe the similarly improved performance, even on Intel Pentium 4.
Among the aforementioned architectures on which we ran our experiments, the 2.4GHz Intel Pentium 4 is unique because it supports a set of SIMD instructions: MMX, SSE and SSE2. We can utilize this feature by passing the enabling options (-xWK) to the Intel Fortran Compiler. First we investigate the effectiveness of turning vectorizer on inside the Intel Compiler. The speedup of the versions of nine point stencil code with vectorizer on compared to those without vectorizer on is shown in Table 2.3. For “orig” and “failsafe”, the average speedup after enabling vectorizer

<table>
<thead>
<tr>
<th>size</th>
<th>orig</th>
<th>failsafe</th>
<th>align</th>
<th>align</th>
<th>lalign</th>
<th>lalign</th>
<th>realign</th>
<th>realign</th>
</tr>
</thead>
<tbody>
<tr>
<td>n=400</td>
<td>1.19</td>
<td>1.22</td>
<td>1.00</td>
<td>1.00</td>
<td>1.59</td>
<td>0.98</td>
<td>1.76</td>
<td>1.00</td>
</tr>
<tr>
<td>n=800</td>
<td>1.19</td>
<td>1.18</td>
<td>1.00</td>
<td>1.00</td>
<td>1.55</td>
<td>1.00</td>
<td>1.65</td>
<td>1.00</td>
</tr>
<tr>
<td>n=1600</td>
<td>1.18</td>
<td>1.17</td>
<td>0.99</td>
<td>1.00</td>
<td>1.56</td>
<td>1.00</td>
<td>1.68</td>
<td>1.00</td>
</tr>
<tr>
<td>average</td>
<td>1.19</td>
<td>1.19</td>
<td>1.00</td>
<td>1.00</td>
<td>1.57</td>
<td>0.99</td>
<td>1.70</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 2.3: Speedup after turning vectorizer on inside Intel compiler for Pentium 4 is 1.19. For “alignmod”, “lalignmod” and “realignmod”, there is no improvement in performance after enabling vectorizer. The compiler report reveals that the inner loops are not vectorized because subscripts with MOD operation are too complex.
For “alignurl”, the inner loop does get vectorized. However, the peeled loop iterations before and after the inner loop (see Figure 2.25) cannot be vectorized due to the low trip count according to the compiler report. We speculate that the overhead of moving data between the sequential and vector instructions killed the benefit of vectorization. For “lalignurl” and “redlalignurl”, since the inner loops are part of perfectly nested loops (see Figure 2.26 and Figure 2.27) and are vectorized, they achieve average speedup of 1.57 and 1.70, respectively. Next, we look at the effectiveness of our loop alignment strategy before and after vectorization. Figure 2.34 (a) shows the normalized running time of “orig”, “orig+vec”, “lalignurl”, “lalignurl+vec”, “redlalignurl”, “redlalignurl+vec”, where suffix “+vec” refers to code compiled with vectorizer turned on. Take “lalignurl” for example, before vectorization, the average speedup of “lalignurl” compared to “orig” is 1.73, while after vectorization, the average speedup of “lalignurl+vec” compared to “orig+vec” is 2.28. We can argue that the reduced memory requirement of “lalignurl” enhanced the effectiveness of vectorization. The speedup relation between these versions are shown in Figure 2.34 (b).

A similar relation also holds for “redlalignurl”.

Figure 2.34: Using vectorizer in Intel compiler for Pentium 4
We have shown that our loop alignment strategy helps improve the memory hierarchy performance by reducing allocated temporary storage size, however, the performance of generated code could be impacted by the backend compiler on a specific architecture. Fortunately, we can generate a spectrum of versions of code, test each one of them and choose the best one. A summary of possible versions for this tuning strategy are shown in Table 2.4.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Transformation options</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce temporary</td>
<td>1) loop alignment 2) lazy loop alignment</td>
</tr>
<tr>
<td></td>
<td>3) reduced lazy loop alignment</td>
</tr>
<tr>
<td>remove array copying</td>
<td>1) loop unrolling 2) rotating array indices</td>
</tr>
<tr>
<td>test unrolling length</td>
<td>1) extra loop unrolling</td>
</tr>
<tr>
<td>software pipelining</td>
<td>1) loop distribution</td>
</tr>
</tbody>
</table>

Table 2.4: A spectrum of versions of code for tuning

2.6.4 Loop Skewing

Although we did not implement loop skewing, we wrote an example and tested hand-coded versions that should be the same as those produced by a loop skewing implementation. First we show the following code in array syntax that performs a simple 4-point 3D stencil computation, similar to the 2D code shown in Section 2.5.3:

\[
\text{grid}(1:N, 2:N+1, 2:N+1) = (\text{grid}(1:N, 1:N, 1:N) \\
+ \text{grid}(1:N, 3:N+2, 1:N) + \text{grid}(1:N, 2:N, 3:N+2)) / 3.0
\]

During scalarization, we keep the loop corresponding to the first dimension innermost as it provides stride-one access and carries no dependence. We experimented on the same four compiler and architecture combinations as above, with four versions of code: (1) the original code in array syntax; (2) the failsafe version, which allocates a \( N \times N \times N \) temporary array; (3) a reduced lazy loop aligned version, which allocates
a $2 \times N \times N$ array; (4) a skewed plus lazy loop aligned version, which allocates a $2 \times N$ array. We refer them as "orig", "failsafe", "redalign" and "skew", respectively. Loop unrolling or rotating array indices are applied to "redalign" and "skew" when profitable. Performance data on grid sizes $120 \times 120 \times 120$, $160 \times 160 \times 160$ and $200 \times 200 \times 200$ are shown in Figure 2.35.

![Figure 2.35: Loop skewing (3D), running time (normalized to original version)](image)

Compared to the original version, we can see from Figure 2.35 that the skewed version clearly demonstrated its benefits of reducing temporary memory size, and it achieved a speedup of 1.60 to 2.06 over the original version on different architectures. Compared to the aligned version, skewed version may run slower on small data sets, as
can be seen on SGI, Compaq and Intel. This could be caused by the code complexity (varied loop bounds) in skewed version and the skewed data access pattern, which affects TLB reuse. However, when the size of data sets increases, the benefits of saving memory pay off and the skewed version performs the best on SUN and Intel, while on SGI and Compaq, performance of the skewed version is comparable to or slightly better than that of the aligned version.

Now we illustrate the penalty of loop skewing if the innermost loop carrying stride-one access is skewed, on the 2D code shown in Section 2.5.3:

\[
\text{grid}(2:N+1, 2:N+1) = (\text{grid}(1:N, 1:N) + \text{grid}(3:N+2, 1:N) + \text{grid}(1:N, 3:N+2)) / 3.0
\]

Similar to the above, we tested on four machines four versions of the code: “orig”, “failsafe”, “redalign” and “skew”. The later three allocate temporary arrays of size \(N \times N, 2 \times N\), and 2, respectively. Performance data on grid sizes \(400 \times 400, 800 \times 800\) and \(1600 \times 1600\) are shown in Figure 2.36. It is clear that the skewed version suffers the penalty of not only the destroyed stride-one spatial reuse, but also the destroyed TLB reuse when the size of the grid increases. Hence we should avoid skewing the innermost loop that carries stride-one memory access.

Having seen the penalty on the 2D code, we want to investigate the profitability of loop skewing: How large should the data size be for the innermost loops nested inside skewed ones? To do that, we separate the sizes of the dimensions of the grid.

\[
\text{grid}(1:M, 2:N+1, 2:X+1) = (\text{grid}(1:N, 1:N, 1:X) \\
+ \text{grid}(1:N, 3:N+2, 1:X) + \text{grid}(1:N, 1:X, 3:X+2)) / 3.0
\]

By varying the innermost dimension size \(M\), we compare the running time of the loop skewed code \((2 \times M\) temporary\) vs. the original code \((K \times N \times M\) temporary\) and the reduced lazy loop aligned code \((2 \times N \times M\) temporary\). Figure 2.37 shows
Figure 2.36: Loop skewing (2D), running time (normalized to original version)

the result of setting $N = K = 200$ while varying $M$ in the set of $\{1, 2, 4, 8, 16, 32, 64, 128, 256\}$. Compared to the original version, loop skewed version performs better when $M$ is bigger than 1. This is also true for different sizes of $N$ and $K$ while keeping $K \times N$ a constant, as shown in Figure 2.38 (a) ($N = 50, K = 800$) and Figure 2.39 (a) ($N = 800, K = 50$). Compared to the reduced lazy loop aligned version, loop skewed version performs better when $M$ is large enough. The value of $M$ at the crossing point that loop skewing is better depends on the architecture and the value of $N$, as shown in Figure 2.37 (b), Figure 2.38 (b) and Figure 2.39 (b). When $N$ is bigger, $M$ at the crossing point is smaller and vice versa. This is because
the temporary array size allocated by reduced lazy loop alignment is $N$ times larger than that allocated by loop skewing.

In summary, for loop skewing to be profitable, it requires innermost dimensions of at least a certain size (combined) not being skewed. The minimal size of innermost dimensions depends on the actual array sizes of skewed dimensions so that reducing the temporary array size can compensate for the penalty caused by skewed data access. This size may be different on different architectures.

(a) skew vs orig (normalized)  
(b) skew vs redialign (normalized)

Figure 2.37: Loop skewing (3D), $N=K=200$, varying size of the innermost dimension (M)

### 2.6.5 Other Applications

Besides stencil computation for solving PDE equations, another application that can benefit from our scalarization strategy is digital image processing. In digital image processing, convolution of an image by a matrix ($3 \times 3$, $5 \times 5$, etc.) can be used to achieve the effects of smoothing, sharpening, edge detection, embossing, cutout, and so on [29]. The matrix used for convolution is usually called a filter or convolution kernel. For example, given an image $A$ as a two dimensional array and a $3 \times 3$ kernel
Figure 2.38: Loop skewing (3D), N=50, K=800, varying size of the innermost dimension (M)

Figure 2.39: Loop skewing (3D), N=800, K=50, varying size of the innermost dimension (M)

c(-1:1, -1:1), each pixel of the resulting image is the weighted average of its eight neighbors and itself: \( A_{i,j} = \sum_{k=-1}^{1} \sum_{l=-1}^{1} c_{k,l} A_{i+k,j+l} \) If we store the resulting image back to the original array, this is exactly the nine-point stencil we have described. The highest grid size in our experiments is 1600 \( \times \) 1600, which only compares to the highest resolution 1600 \( \times \) 1200 of a 2.0 megapixel digital camera. Nowadays we see the
emerging products of 5.0 megapixel and more, which makes our scalarization strategy more beneficial to improve the memory hierarchy performance of these digital filtering algorithms.

We attempted to find other applications to illustrate the benefits of our loop alignment and loop skewing strategy, but without marked success. We see two reasons for this. First, most iterative PDE solvers need to check convergence after each iteration, therefore a copy of the old array has to be kept for computing the difference. Second, we know that application programmers are aware that, with most compilers, allocating temporary arrays causes problems in cache. To avoid those problems they probably choose to write hand optimized code in loop nests instead of using array syntax. To put it another way, more widespread use of the scalarization techniques presented in this dissertation might lead programmers to write code that are cleaner and easier to read in Fortran 90.

The strategies that we have developed in this chapter are so effective that it suggests that they might be applied to code that is not written in array syntax. The approach would be to use advanced vectorization strategies, such as those described in the book by Allen and Kennedy [4], to produce a canonical array version of a loop nest. Then the scalarization algorithms described here could be applied to minimize temporary storage. However, it remains to be seen whether this strategy would produce any real benefits for legacy codes.

2.7 Summary

We have presented novel applications of two known compiler strategies, loop alignment and loop skewing, to the problem of scalarizing array statements. The goal of
these two approaches is to eliminate or reduce the need for temporary array storage, thus improving the memory hierarchy performance on a single scalar processor. We also have shown that, for stencil calculations, there is an algorithm for applying these two transformations to produce a scalarization that has asymptotically optimal (minimal) temporary storage requirement. Another benefit of minimizing memory footprint size is that it enables array computations of a larger size on a given amount of memory.

We have implemented loop alignment and our experiments have demonstrated a dramatic performance gain on large data sets. Although we did not implement loop skewing due to the complexity of analysis, our hand-coded examples show that loop skewing can achieve performance gains similar to or better than loop alignment. In particular, loop skewing should be tried before loop alignment. If it succeeds, no temporary is needed for participating loops. If it fails, loop alignment should be used. Thus, the combination of loop skewing and loop alignment can only produce better results than alignment by itself on large data sets. Coupled with loop unrolling or rotating array indices whenever possible, the overhead of register copying or array copying to support loop alignment can be eliminated.

We also addressed the interferences between scalarization and backend compilation. We argued that the contributing factors to performance includes not only the memory footprint size that is what our scalarization strategy tries to minimize, but also many other factors such as quality of compiled code, memory locality and so on. In this chapter, we identified three issues: (1) Lazy loop alignment may perform better than loop alignment and reduced lazy loop alignment even though it requires a slightly larger temporary array size. (2) Scalarization should preserve stride-one access in the innermost loop. (3) The profitability of loop skewing depends on the
actual array sizes. To make the choice of whether to use loop alignment, lazy loop alignment or reduced lazy loop alignment, whether to use loop skewing or one loop alignment variant, whether to use loop unrolling or rotating array indices, and which skewing formula to use, we could do offline performance tuning or run-time adaptation. Automating this tuning process with an iterative compilation framework [32] is feasible.

Although our experiments were with Fortran 90, the scalarization techniques described here can be applied to other languages with array syntax features. For example, the Overture library [11] provides multi-dimensional arrays as C++ classes and operations that treat array objects as a whole, similar to array sections in Fortran 90. And Matlab, an extremely popular language within science and engineering uses array assignment statements.

The scalarization techniques described in this chapter work for a scalar processor, which is a special case of vector processors in the sense that the vector length is 1. When the length of the array assignments exceeds that of the vector on the underlying architecture, we have to break the array assignments into a sequence of the vector operations. Therefore, we still need to preserve the dependence — the same issue we had with the scalar machine. The only difference is the vector length. The results presented in this chapter should extend to general vector processors. Our experiments on the Pentium 4 indicate that the reduced memory requirement provided by our scalarization techniques can enhance the efficiency of an underlying vector architecture. We will discuss the scalarization for vector processors in Chapter 3.

Both loop alignment and loop skewing handle the case where naively scalarized loop nest carries both true dependences and antidependences. By reducing the size of temporaries, these two techniques may prohibit the parallelism that can be achieved
by using the failsafe method. How to parallelize array assignments while maintaining efficient memory hierarchy performance [48] will be addressed in Chapter 4, when we compile for a heterogeneous multi-core CELL processor.
Chapter 3

Compiling for Short Vector Processors

In this chapter, we extend the scalarization strategy for scalar processors presented in Chapter 2 to vector processors. In particular, we are interested in the short vector units existed in modern microprocessors, such as SSE instructions on x86 processors and AltiVec instructions on PowerPC processors. When compiling array statements for high performance using these vector units, one interesting problem is that the temporary memory reduction strategy may prevent a loop from being vectorized. Another problem is related to the data alignment constraints associated with the vector data accesses—aligned vector data accesses are much faster than unaligned ones. Therefore, our extended scalarization strategy should address these two problems accordingly.

In what follows, Section 3.1 introduces the short vector processors that we target and the two problems that may prevent effective scalarization, Section 3.2 reviews related work, Section 3.3 and Section 3.4 presents our solutions for the two problems
accordingly, Section 3.5 shows the experimental results, and Section 3.6 summarizes this chapter.

3.1 Introduction

Modern scalar microprocessors are often equipped with a few vector function units for SIMD/vector executions. Examples are SSE instructions on latest x86 and x86-64 processors from Intel and AMD, and AltiVec instructions on PowerPC processors from IBM and Motorola. Initially designed for speeding up multimedia applications, these vector instruction sets can also speed up scientific applications when fully utilized.

Compared to traditional vector processors, the aforementioned short vector processors have the following architecture constraints:

- **Short vector length** The vector length is relatively short, 16 bytes, which translate to four integers, four single precision floating points, or two double precision floating points. On PowerPC, double precision is not supported by AltiVec. This constraint caps the amount of speedup we can get from these vector units.

- **Contiguous memory access** Vector loads and stores can only access contiguous regions in memory; i.e., no strided vector data accesses are supported. This constraint limits the amount of programs that we can vectorize for these vector units.

- **Data alignment** Vector accesses to data aligned on 16-byte boundaries are faster than those to unaligned ones. For instance, PowerPC AltiVec architecture requires that vector data load/store instructions operate on the 16-byte
boundaries. In which case, a unaligned vector load will have to be implemented as two aligned vector loads followed by a vector merge operation, as shown in Figure 3.1 (a). Although SSE instructions on latest x86 architectures support both aligned and unaligned vector data accesses, unaligned version takes longer than the corresponding aligned one. Figure 3.1 (b) shows the running time in seconds of two versions of a program, one with aligned vector load instruction, the other with unaligned vector load instruction even though the data remain aligned. We can see a huge performance penalty associated with the unaligned loads. Therefore, this data alignment constraint require that we align as many data accesses as possible in the vectorized code.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>800</th>
<th>1600</th>
<th>3200</th>
<th>6400</th>
</tr>
</thead>
<tbody>
<tr>
<td>load_ps</td>
<td>0.75</td>
<td>1.42</td>
<td>3.21</td>
<td>6.37</td>
</tr>
<tr>
<td>load_ups</td>
<td>1.12</td>
<td>2.21</td>
<td>4.69</td>
<td>9.38</td>
</tr>
</tbody>
</table>

(a) PowerPC Altivec    (b) Intel Pentium IV SSE

Figure 3.1: Performance penalty for unaligned data accesses

Nevertheless, when a program is compiled to utilize these vector instructions, its performance can be greatly improved. Though of limited vector processing power compared to traditional vector machines, these short vector processors are available almost everywhere.

In this chapter, we extend the scalarization strategy for scalar processors to the short vector processors. The goal is to obtain high performance for the scalarized code. As discussed above, it requires not only sufficiently identifying fine-grain parallelism in user applications, but also ensuring proper data alignment for vectorized data
accesses. Thus there are two technical problems for us to solve:

- **Vectorization** The scalars generated by the temporary reduction strategy we developed for scalar processors in Chapter 2 may prevent the innermost loop from being vectorized. As a solution, we will show how to extend the scalarization strategy to fully utilize these short vector units, while at the same time, preserving the “fetch before store” semantics and keeping the temporary memory requirement minimized.

- **Data alignment** It is desirable to have as many array references aligned as possible in the scalarized code with vector operations. For this problem, we present a combined loop alignment algorithm that takes both temporary memory reduction and aligned data access into consideration.

We also look at a variation of scalar replacement for vectors that may further improve the memory hierarchy performance on these short vector processors.

### 3.2 Related Work

Larsen and Amarasinghe [36] referred to SIMD capability in the short vector processors as *superword-level parallelism (SLP)*. They developed an algorithm that uses loop unrolling and statement regrouping within a basic block to identify adjacent memory references that can be subsequently replaced with SLP vectors. In our work, we start from vector code in array syntax and scalarize them onto short vector machines. We are interested in cases that the scalarized code cannot be directly unrolled and regrouped.
Another compiler technique to take advantage of SIMD units is loop-based vectorization. Initially developed for conventional vector machines, loop-based vectorization has been well-known for over 20 years. For short vector processors, this technique is implemented in many compilers, including Intel [6], VAST/Altivec [18], IBM XL [21], and GNU [41]. Our approach fits into this category because we also rely on the dependence information to infer parallelism.

Shin et al [49] introduced a variation of scalar replacement [14] for vector references, which they define as *superword-level locality (SLL)*, to improve the reuse of values in vector registers. Their algorithm uses unroll-and-jam to expose spatial reuse opportunities. In our work, we only focus on the vector register reuse in the innermost loop itself without unroll-and-jam because our main purpose is to discuss the data alignment issue. We do present the performance study of the vectorized scalar replacement.

Loop peeling is a simple but efficient technique to align an array reference on the first iteration of a loop. It has been described and used in many compiler implementations [6, 21]. However, it cannot align all array references if they do not have the same memory alignment size (the number of iterations needs to be peeled in order to make an array reference aligned). Eichenberger et al [21] described an algorithm that efficiently aligns all vectorized array references in a software pipelined fashion to minimize the number of vector loads/stores and vector merge operations. Their approach is similar to the one in VAST/Altivec compiler [18] as suggested by coding examples on VAST website. The key idea is to treat an unaligned array reference in a loop as a data stream rather than individual references at each iteration. While in our work, we identified additional opportunities for data alignment that can be exposed by loop alignment, and our input is vector code that include cases that the
corresponding scalar version cannot be directly unrolled for vectorization. Combined with their algorithm, our strategy can also help reduce register pressure in certain cases.

Fraboulet et al [24] used loop alignment for reducing the number of iterations between two array references in different statements that have data reuses. Their algorithm is target at embedded systems and they do not consider vector execution and data alignment issue.

Allen and Kennedy[3] directly scalarize array statements to the vector length. This is the closest work to ours. In their paper, input prefetching is used to preserve the array syntax semantics and the performance study is lacking. While in our study, we take a similar direct scalarization approach except that we use loop alignment instead of input prefetching. Our approach can not only preserve array syntax semantics with a small temporary memory allocation, but also improve the data alignment. We will evaluate both approaches on the target architectures.

3.3 Scalarization Strategy for Short Vector Processors

Short vector processors only support vectorized memory accesses to consecutive regions; i.e., there are no vectorized stride accesses that are often found on traditional vector machines. Therefore, we only consider the array statements whose innermost dimension can be vectorized and it suffices to only study the innermost dimension. Scalarization of outer dimensions can be handled by the previous scalarization framework for scalar processors. A naively scalarized loop is the result of enumerating the
elements of array sections with a loop. It is correct only if it does not carry any true dependences [4], in which case vectorization is also straightforward. Here we give a scalar and vector code generation example for array assignment: $A(1:N-1) = A(1:N-1) + A(2:N)$ in Figure 3.2, where $VL$ is the vector length and $N - 1 > VL$.

```
!! code for scalar machine:
DO I = 1, N-1
  A(I) = A(I) + A(I+1)
ENDDO

!! code for vector machine:
DO I = 1, N-1-(VL-1), VL
  A(I:I+VL-1) = A(I:I+VL-1) + A(I+1:I+VL)
ENDDO
DO I = 1+(N-1)/VL+VL, K-1
  A(I) = A(I) + A(I+1)
ENDDO
```

Figure 3.2: Simple scalarization example

However, when the naively scalarized code does not preserve the original array syntax semantics, we need to allocate temporary storage. There are usually three approaches for scalar machines: 1) failsafe method, 2) input prefetching, and 3) loop alignment. The first method allocates a full size temporary array, while the later two utilize temporary arrays of smaller sizes to improve the performance. In one dimensional case, the smaller temporary arrays can often be replaced with a number of scalars. However, when trying to fully utilize the vector capabilities of our target short vector machines, except for the failsafe method, the later two cannot be directly vectorized. We explain with an array statement: $A(2:N-1) = A(1:N-2) + A(3:N)$ in Figure 3.3.

From Figure 3.3, we can see that the naively scalarized version is wrong because it has a different meaning from the original array assignment. To reduce temporary array size, input prefetching prefetches the read references before any write into that memory location, while loop alignment temporarily hold a write reference until all
reads from that memory location are finished. These two versions allocate temporaries of size 2 compared to size $N-2$ in the failsafe version, however, they can not be directly vectorized, nor can they be unrolled and regrouped to expose adjacent memory accesses. Instead of vectorizing the scalar version, Allen and Kennedy [3] directly scalarize the original array assignment to the vector form using input prefetching on vectors. Here we can take a similar approach for loop alignment by expanding the allocated temporary storage, as shown in Figure 3.4.

From Figure 3.4, we can see that (f) allocates a temporary array of size $d + 1 + (VL - 1)$, where $d$ is the alignment distance ($d = 1$ in this case). Notice that the expanded part of the temporary array is used only in the main loop body. If we can use the underlying vector registers directly, we shall rewrite the code in Figure 3.5, where $TV1$ and $TV2$ are arrays of length $VL$ that we hope can be mapped to vector registers. EOSHIFT and MERGE are Fortran 90 intrinsic functions: EOSHIFT shifts the elements inside a vector, while MERGE selects between elements in two vectors into a result vector based on a mask.

Final notes about the loop alignment algorithm are: 1) the scalar copy and vector
TVL = A(1:VL)
DO I = 1, N-2 -(VL-1), VL
   TV2 = TV1 + A(I+2:I+VL+1)
   TV1 = A(I+VL:I+VL+VL-1)
   A(I+1:1+VL) = TV2
ENDDO
T1 = TV1(1)
DO I = 1+((N-2)/VL)*VL, N-2
   T2 = A(I) + A(I+2)
   T1 = A(I+1)
   A(I+1) = T2
ENDDO

(e) input prefetching, vectorized
T(1) = A(1) + A(3)
DO I = 1, N-3 -(VL-1), VL
   T2 = A(I+1:I+VL) + A(I+3:I+VL+2)
   A(I+1:I+VL) = T(1:VL)
   T(1) = T(VL+1)
ENDDO
DO I = 1+((N-3)/VL)*VL, N-3
   T(2) = A(I+1) + A(I+3)
   A(I+1) = T(1)
   T(1) = T(2)
ENDDO
A(N-1) = T(1)

(f) loop alignment, vectorized

Figure 3.4: Vectorized scalarization example

copy in Figure 3.3 (d) and Figure 3.5 will be eliminated by loop unrolling [4] during the code generation. They are kept in these examples only for convenience of reading and understanding. 2) If the loop alignment distance (the number of iterations peeled) is greater than VL, then we need more vector registers to temporarily hold the computation result.

3.4 Data Alignment on Short Vector Processors

On the short vector machines which we target, data alignment has a significant memory hierarchy performance implication. Our optimization goal is to align as many data accesses as possible in our generated code to the required boundary. In what follows,
!! LOGICAL MASK(VL) = (/..T., .F., ..., .F./)
T1 = A(I) + A(I+3)
TV1(VL) = T1
DO I = 1, N-3 - (VL-1), VL
TV2 = A(I+1:I+VL) + A(I+3:I+VL+2)
A(I+1:I+VL) = MERGE(BISHEPT(TV1, VL-1), &
                  BISHEPT(TV2, -1), MASK)
TV1 = TV2
ENDDO
T1 = TV1(VL)
DO I = 1+((N-3)/VL)*VL, N-3
T2 = A(I+1) + A(I+3)
A(I+1) = T1
T1 = T2
ENDDO
A(N-1) = T1

Figure 3.5: Scalarization example (g): vector register allocated

we give details about how to pick a unified data alignment pivot from write access
and read accesses, how to combine vectorized scalar replacement in this process, and
the updated loop alignment algorithm for short vector processors.

Data alignment pivot  The array references in the scalarized code are not
exactly vector length apart from each other. Thus we need to pick one of them as the
data alignment pivot, the rest will have either aligned or unaligned access accordingly.
In the context of scalarization, we can choose the left hand side array reference (write
access) as the pivot. Later we will show that this choice does not prohibit an optimal
alignment for the read accesses and hence our total strategy will achieve an optimal
alignment. To align the chosen write access, we simply need to peel a certain number
of loop iterations at the beginning. This number is in the range of [0, VL-1] and can
be determined dynamically at run time by inserting an address calculating formula at
compile time. We call this number data alignment size, which of reference A(I) can be
computed by \textit{AlignSize}(A(I)) = (\textit{base} - I) (PMOD \textit{VL}), where \textit{A(base)} is data aligned
and \textit{PMOD} only returns a non-negative number (we cannot peel a negative number of
iterations): \( x \ (\text{PMOD} \textit{VL}) = ((\textit{VL} + (x \mod \textit{VL})) \mod \textit{VL}) \). For the example code
shown in Figure 3.5, suppose that $A(1)$ is data aligned, to properly align the write access $A(I+1:I+VL)$, we need to peel $VL - 1$ iterations of the loop at the front.

**Pivot for read accesses** Now we consider how to maximize the benefits of read access alignment. For all read accesses, their data alignment sizes range from 0 to $VL - 1$ and therefore can be partitioned into $VL$ equivalent classes. A simple solution is to choose one class that has most array references and make one reference in it data alignment pivot. For the example code in Figure 3.5, suppose $A(1)$ is data aligned, the data alignment sizes for read accesses $A(I+1:I+VL)$ and $A(I+3:I+VL+2)$ are $VL - 1$ and $VL - 3$, respectively (assuming that $VL > 3$). Thus we can pick either one as the pivot. Later in the paper it becomes clear that $A(I+1:I+VL)$ is conformable to the write pivot, while $A(I+3:I+VL+2)$ is not and requires additional processing.

**Vectorized scalar replacement** The memory hierarchy performance of read accesses can be further improved by *scalar replacement* [14]. Scalar replacement reduces memory traffic by identifying loop independent and loop carried temporal reuses among array references and converting them into register reuses. Basically, the algorithm first partitions array references into reference groups based on temporal reuse. After that for each group, a leader is identified and used to bring the value from memory into a scalar variable, and the rest references in the group will be rewritten as scalar variables that copy values from the group leader. If the new scalar variables are register allocated properly by the backend compiler, the original memory access pattern will be replaced by accessing only the reference group leaders into the memory along with a few register copies that can often be eliminated by loop unrolling. For the example code in Figure 3.3 (a), $A(I+3)$ is the leader of the group that also includes $A(I+1)$. 
In the context of scalarization for short vector machines, we need to extend regular scalar replacement into a variant as *vectorized scalar replacement*. The extension is straightforward: reference group leaders fetch value into vector variables, the rest are converted into vector variables who get value from vector copies and vector merges. Now with vectorized scalar replacement in mind, the best data alignment will only consider the reference group leaders since they are the only ones that access memory. The data alignment pivot will be chosen from the data alignment equivalent class that has the most reference group leaders. For the code shown in Figure 3.5, $A(I+3:I+VL+2)$ is the leader of group that also includes $A(I+1:I+VL)$, and thus is chosen to be the data alignment pivot for read accesses. Suppose that $A(1)$ is data aligned, the data alignment size for $A(I+3:I+VL+2)$ is $VL - 3$.

**Conform two pivots** Now we have chosen two data alignment pivots, one for the write access, the other for the read accesses. But how do we satisfy both alignment sizes in one loop if they are not equal? We have to equalize these two sizes. The solution is to increase the loop alignment distance and re-do the loop alignment. Note that the old alignment distance is only the minimum distance that guarantees the correctness of the generated code. The new alignment distance is:

$$new\text{AlignDis} = min\text{AlignDis} + (\text{AlignSize(write pivot)} - \text{AlignSize(read pivot)}) \mod VL)$$
$$= min\text{AlignDis} + ((\text{Subscript(read pivot)} - \text{Subscript(write pivot)}) \mod VL)$$

The second equal sign holds when the read pivot and write pivot are references to the same array, which is the case we are considering in this dissertation.

With the new alignment distance, essentially we are shifting the read pivot so that it aligns properly with the write pivot. For our example code, suppose that $VL = 4$ and $A(1)$ is data aligned, the alignment distance increases from 1 to 3 and Figure 3.6
gives the final result after vectorized scalar replacement, optimal data alignment and register copy removal.

!! vectors: RV1, RV2, TV1, TV2, TV3
LOGICAL MASK(4) = (/ .T., .T., .T., .F./)
LOGICAL RMASK(4) = (/ .T., .T., .F., .F./)

T(1:3) = A(1:3) + A(3:5)
!! loop peeling for data alignment
!! PEEL: determined at runtime
!! PEEL=3 if A(i) aligned
PEEL = AlignSize(A(2))
DO I = 1, PEEL
  T(4) = A(I+3) + A(I+5)
  A(I+1) = T(1)
  T(1:3) = T(2:4)
ENDDO
TV1(2:4) = T(1:3)
!! prefetch for vectorized scalar replacement
RV1 = A(PEEL*2:PEEL+6)
DO I = PEEL+1, N-8-7, 4+2
  RV1 = A(I+5:I+8)  !! aligned vector fetch
  RV2 = A(I+5:I+8)  !! aligned vector fetch
  RV1 = MERGE(EOSHIFT(RV1,2), &
               EOSHIFT(RV2,-2), RMASK)
  TV2 = RV1 + RV2
  TV3 = MERGE(EOSHIFT(TV3,1), &
               EOSHIFT(TV3,-3), MASK)
  A(I+1:I+4) = TV3  !! aligned vector store
  A(I+1:I+4) = TV3  !! aligned vector store
  RV1 = RV2  !! vector copy removal
  TV1 = TV2  !! by loop unrolling
  RV1 = A(I+9:I+12)
  RV2 = MERGE(EOSHIFT(RV2,2), &
               EOSHIFT(RV1,-2), RMASK)
  TV1 = RV2 + RV1
  TV3 = MERGE(EOSHIFT(TV2,1), &
               EOSHIFT(TV1,-5), MASK)
  A(I+5:I+8) = TV3
ENDDO

T(1:3) = TV1(2:4)
DO I = PEEL+1 + ((N-8-PEEL)/8) * 8, N-5
  T(4) = A(I+3) + A(I+5)
  A(I+1) = T(1)
  T(1:3) = T(2:4)
ENDDO
A(N-3:N-1) = T(1:3)

Figure 3.6: Scalarization example (h): vectorized scalar replacement and data alignment

Of course, this new alignment distance considering data alignment can be determined in the process of loop alignment transformation, using the original read pivot from the naively scalarized version:
\[ \text{newAlignDis} = \text{minAlignDis} + \left( (\text{Subscript(ori read pivot)} - \text{Subscript(write pivot)}) + \text{minAlignDis} \right) \text{ (PMOD VL)} \]

Recall that we choose the read pivot from the data alignment equivalent class that has the most reference group leaders. If there are multiple such classes, we can break the tie by selecting the one with minimum \text{newAlignDis}.

**Array padding** Even though we target the innermost dimension for vectorization, however, conforming two pivots in a two-dimensional array generally cannot be statically determined. For example, \( A(I, J) \) may or may not be aligned on the same boundary as \( A(I, J+1) \), if the array is simply declared as \( A(N, M) \) (assuming column major as in Fortran). To make sure that they are always aligned, one solution is array padding: declare \( A(N + ((VL-(N \text{ MOD } VL)) \text{ MOD } VL), M) \) instead. This way, we only need to consider the array reference subscript in the innermost dimension for calculating the relative data alignment size.

**Updated loop alignment algorithm** Putting it all together, we give the combined loop alignment algorithm for scalarization on short vector machines considering data alignment in Figure 3.7.

```plaintext
procedure scalarize(S)
    // S: one dimensional array assignment statement
    naively scalarize S into a loop
    compute the minimum loop alignment distance
    determine the write pivot and read pivot
    update the loop alignment distance
    align loop for correctness and data alignment
    peel loop for data alignment
    generate vectorized code
    unroll loop for register copy removal
```

Figure 3.7: Scalarization algorithm considering data alignment
3.5 Experiments

We conducted our experiments on two types of architectures: a Intel Pentium 4 (2.4GHz) equipped with SSE extensions, and an Apple PowerPC G4 (1.33GHz) and a G5 (2.5GHz) with Altivec. Pentium 4 has 8 128-bit vector registers, a 8K L1 cache and a 512K L2 cache, while both G4 and G5 have 32 128-bit vector registers, 32K L1 cache and 512K L2 cache. Two code written in Fortran 90 that we tested are a one-dimensional three point stencil (3point) similar to our example above:

\[ A(2:N-1) = \frac{A(1:N-2) + A(3:N)}{2} \]

and a two-dimensional five point stencil (5point):

\[
\begin{align*}
\text{DO} & \text{ J = 2, N-1} \\
\end{align*}
\]

The arrays are declared of type 32-bit floating point. Thus the vector length is 4 for both architectures. These code are iterated for a certain amount of iterations for observable running time. The tested programs also have an initialization phase at the beginning and a checksum computation at the end. For 3point, the tested problem sizes (n) are 200, 400, 800, 1600, 3200, 6400, 12800, 25600 and 51200, while for 5point, the tested problem sizes (n) are 50, 75, 100, 150, 200, 300, and 400. However, due to the space limit, we can only show some of them. The average speedup is actually computed on all tested problem sizes.

In the following, for each architecture, we compare the performance among the original code in array syntax, the failsafe code that allocates a full size temporary array, the code transformed by the input prefetching strategy, and the code by our loop alignment strategy, all compiled by the available native compiler. We also evaluate
the data alignment strategies presented in Section 3.4. For convenience, we use the
following abbreviations and suffixes in Table 3.1 for reference of different versions of
the programs.

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig</td>
<td>original program in array syntax</td>
</tr>
<tr>
<td>Fsafe</td>
<td>scalarized failsafe version</td>
</tr>
<tr>
<td>Pfetch</td>
<td>scalarized input prefetching version</td>
</tr>
<tr>
<td>Lalign</td>
<td>scalarized loop alignment version</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+V</td>
<td>vectorized</td>
</tr>
<tr>
<td>+A</td>
<td>data aligned</td>
</tr>
<tr>
<td>+R</td>
<td>vectorized scalar replacement</td>
</tr>
<tr>
<td>+P</td>
<td>array padding</td>
</tr>
</tbody>
</table>

Table 3.1: Versions of the programs

3.5.1  **SSE on Pentium 4**

We used Intel Fortran compiler version 7.1 for this experiment. It has an automatic
vectorization [6] option for utilizing the underlying SSE instructions. With the option
turned off and on, we tested four versions of programs on different data sizes. The
running time and average speedup compared to a baseline version (Orig) for 3point
is shown in Table 3.2.

The Lalign+V version is the one with our proposed strategy. However, the Intel
compiler failed to recognize a size 4 floating point array as a vector, even though it has
always been used as a whole identity in the loop. Lalign+V also suffered from the bad
implementation of intrinsic functions MERGE and EOSHIFT. Nevertheless, we can
see from the non-vectorized version, the average speedup of Lalign over Orig is 1.33.
Pfetch performs equally well as Lalign, and they are even faster than Orig+V. When
<table>
<thead>
<tr>
<th>Size(n)</th>
<th>200</th>
<th>800</th>
<th>3200</th>
<th>12800</th>
<th>51200</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig</td>
<td>0.41</td>
<td>1.76</td>
<td>6.64</td>
<td>28.04</td>
<td>249.63</td>
<td>baseline</td>
</tr>
<tr>
<td>Fsafe</td>
<td>0.45</td>
<td>1.78</td>
<td>7.19</td>
<td>28.77</td>
<td>220.61</td>
<td>0.98</td>
</tr>
<tr>
<td>Pfetch</td>
<td>0.37</td>
<td>1.49</td>
<td>5.65</td>
<td>22.58</td>
<td>90.25</td>
<td>1.33</td>
</tr>
<tr>
<td>Lalign</td>
<td>0.35</td>
<td>1.41</td>
<td>5.71</td>
<td>22.95</td>
<td>104.70</td>
<td>1.33</td>
</tr>
<tr>
<td>Orig+V</td>
<td>0.40</td>
<td>1.50</td>
<td>5.86</td>
<td>23.34</td>
<td>205.94</td>
<td>1.15</td>
</tr>
<tr>
<td>Fsafe+V</td>
<td>0.39</td>
<td>1.50</td>
<td>5.84</td>
<td>23.28</td>
<td>180.77</td>
<td>1.17</td>
</tr>
<tr>
<td>Pfetch+V</td>
<td>0.98</td>
<td>3.98</td>
<td>15.76</td>
<td>63.16</td>
<td>253.37</td>
<td></td>
</tr>
<tr>
<td>Lalign+V</td>
<td>20.60</td>
<td>84.60</td>
<td>345.49</td>
<td>1467.34</td>
<td>5638.74</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Running time of 3point on Pentium 4

the array size becomes larger, Pfetch and Lalign clearly demonstrated the benefits of reducing the temporary storage. We observe similar behavior for 5point.

To investigate the effectiveness of our strategy, we need a compilation environment that can map the vector array onto the underlying vector architecture, as we intended. Fortran failed us as we have seen here. Fortunately, SSE has been implemented as a language extension (intrinsics) in C/C++ and supported by both Intel and GNU C/C++ compilers. Therefore, we implemented a procedure-outlining strategy that converts Fortran loops into C procedures and rewrites the computation using the provided vector data type and vector intrinsics. In this environment, we can also study the effectiveness of data alignment and vectorized scalar replacement. The results are shown in Table 3.3 and Table 3.4. Note that cFsafe+V is vectorized automatically by Intel C/C++ compiler version 7.1. The optimization level is set to O3 for all compilers.

By directly accessing the underlying short vector feature, we can see that our scalarization strategy work quite well. In order to understand the memory hierarchy performance, we also collect the number of vector load and vector store instructions, either aligned or unaligned, from the instruction trace of 3point on a problem size of 200, as shown in Table 3.5.
Table 3.3: Running time of 3point (C implementation) on Intel Pentium 4

<table>
<thead>
<tr>
<th>Size(n)</th>
<th>200</th>
<th>800</th>
<th>3200</th>
<th>12800</th>
<th>51200</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>cFetch</td>
<td>0.42</td>
<td>1.45</td>
<td>6.05</td>
<td>24.01</td>
<td>158.31</td>
<td>baseline</td>
</tr>
<tr>
<td>cFetch+V</td>
<td>0.27</td>
<td>1.07</td>
<td>4.50</td>
<td>17.94</td>
<td>71.07</td>
<td>1.48</td>
</tr>
<tr>
<td>cFetch+VA</td>
<td>0.44</td>
<td>1.71</td>
<td>6.76</td>
<td>26.83</td>
<td>107.67</td>
<td>0.96</td>
</tr>
<tr>
<td>cFetch+VAR</td>
<td>0.39</td>
<td>1.58</td>
<td>6.35</td>
<td>25.64</td>
<td>104.27</td>
<td>1.02</td>
</tr>
<tr>
<td>cAlign</td>
<td>0.27</td>
<td>1.03</td>
<td>4.27</td>
<td>17.14</td>
<td>68.25</td>
<td>1.56</td>
</tr>
<tr>
<td>cAlign+V</td>
<td>0.25</td>
<td>0.94</td>
<td>4.06</td>
<td>15.96</td>
<td>65.69</td>
<td>1.64</td>
</tr>
<tr>
<td>cAlign+VA</td>
<td>0.60</td>
<td>2.40</td>
<td>14.64</td>
<td>58.99</td>
<td>228.84</td>
<td>0.55</td>
</tr>
<tr>
<td>cAlign+VAR</td>
<td>0.21</td>
<td>0.73</td>
<td>3.15</td>
<td>12.47</td>
<td>49.93</td>
<td>2.11</td>
</tr>
<tr>
<td>cAlign+VAR</td>
<td>0.24</td>
<td>0.75</td>
<td>3.21</td>
<td>12.67</td>
<td>50.32</td>
<td>2.04</td>
</tr>
</tbody>
</table>

Table 3.4: Running time of 5point (C implementation) on Intel Pentium 4

<table>
<thead>
<tr>
<th>Size(n)</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>cFetch</td>
<td>1.00</td>
<td>5.18</td>
<td>18.12</td>
<td>87.24</td>
<td>baseline</td>
</tr>
<tr>
<td>cFetch+V</td>
<td>0.67</td>
<td>2.68</td>
<td>11.21</td>
<td>50.36</td>
<td>1.57</td>
</tr>
<tr>
<td>cFetch+VA</td>
<td>1.08</td>
<td>3.39</td>
<td>12.38</td>
<td>74.56</td>
<td>1.22</td>
</tr>
<tr>
<td>cFetch+VAR</td>
<td>0.75</td>
<td>2.93</td>
<td>10.80</td>
<td>47.15</td>
<td>1.52</td>
</tr>
<tr>
<td>cFetch+VPA</td>
<td>0.66</td>
<td>2.33</td>
<td>8.58</td>
<td>42.94</td>
<td>1.85</td>
</tr>
<tr>
<td>cFetch+VPAR</td>
<td>0.66</td>
<td>2.50</td>
<td>9.03</td>
<td>44.48</td>
<td>1.80</td>
</tr>
<tr>
<td>cAlign</td>
<td>0.63</td>
<td>2.25</td>
<td>8.27</td>
<td>41.90</td>
<td>1.99</td>
</tr>
<tr>
<td>cAlign+V</td>
<td>0.67</td>
<td>2.67</td>
<td>11.17</td>
<td>48.68</td>
<td>1.59</td>
</tr>
<tr>
<td>cAlign+VA</td>
<td>0.97</td>
<td>4.54</td>
<td>17.32</td>
<td>76.51</td>
<td>1.03</td>
</tr>
<tr>
<td>cAlign+VAR</td>
<td>0.69</td>
<td>2.66</td>
<td>10.57</td>
<td>46.40</td>
<td>1.71</td>
</tr>
<tr>
<td>cAlign+VPA</td>
<td>0.82</td>
<td>2.54</td>
<td>9.20</td>
<td>42.46</td>
<td>1.75</td>
</tr>
<tr>
<td>cAlign+VPAR</td>
<td>0.65</td>
<td>2.28</td>
<td>8.57</td>
<td>42.16</td>
<td>1.90</td>
</tr>
<tr>
<td>cAlign+VPAR</td>
<td>0.58</td>
<td>2.21</td>
<td>7.59</td>
<td>39.89</td>
<td>2.06</td>
</tr>
</tbody>
</table>

In the following, we summarize different contributing factors that lead to the performance improvements:

1. **Data alignment** The data aligned version (+VA) is much faster than the unaligned one (+V) on almost all occasions. Table 3.5 shows that most of unaligned vector memory accesses are converted to aligned ones after the data alignment. The data alignment strategy for cFetch+VA is to always pick the write access as the alignment pivot, while the strategy for cAlign+VA is to use the combined loop alignment algorithm described in Section 3.4. In fact,
Table 3.5: Vector load and store instructions of 3point (C implementation) on Intel Pentium 4

both cPfetch+V and cLalign+V without data alignment run slower than the corresponding non-vectorized version.

2. Vectorized scalar replacement Exploiting data reuse through vector registers speeds up programs universally except the cLalign+VA version for 3point, in which case cLalign+VAR is about 4% slower. We believe that two reasons contributed to this phenomenon: 1) the vector load from L1 cache is relatively fast on Pentium 4, and 2) the vectorized scalar replacement requires vector shuffling operations, sometimes even two shuffles for one eliminated vector load operation. We can see from Table 3.5 that the number of vector load instructions are halved after vectorized scalar replacement.

3. Array padding For 5point, array padding makes references $A(I, J-1), A(I, J)$ and $A(I, J+1)$ aligned on the same boundary. Both cPfetch+VPA and cLalign+VPA can exploit this property and generate aligned vector loads. This clearly translates into the reduction of running time.

4. Overall performance In summary, code generated by our scalarization strat-
egy for short vector architecture achieves an integer speedup over the one automatically vectorized by the native compiler, and also performs much better than the Fortran counterparts and slightly better than the input prefetching strategy. This speedup becomes larger for larger problem sizes due to the reduction of temporary memory allocation, as shown in Figure 3.8.

Figure 3.8: Running time of 3point and 5point on Pentium 4

3.5.2 AltiVec on PowerPC G4/G5

On PowerPC G4 and G5, we tested the two stencil code with the IBM XL Fortran compiler version 8.1. We found that the vectorization capability is provided as an add-on software from VAST/AltiVec by Crescent Bay Software [18]. While revising this paper, we obtained a copy of VAST/AltiVec, which rewrites a vectorized Fortran loop in a C function that explicitly uses AltiVec interface. However, it does not translate the code with our scalarization strategy into C, namely, arrays of size 4, EOSHIFT and MERGE intrinsics. Nevertheless, we present the results of 3point in Table 3.6 to show the performance advantage of Pfetch and Lalign compared to the native compiler behavior on Orig.
<table>
<thead>
<tr>
<th>Size(n)</th>
<th>200</th>
<th>800</th>
<th>3200</th>
<th>12800</th>
<th>51200</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>G4:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orig</td>
<td>0.30</td>
<td>1.12</td>
<td>4.33</td>
<td>30.95</td>
<td>248.63</td>
<td>baseline</td>
</tr>
<tr>
<td>Fsafe</td>
<td>0.30</td>
<td>1.13</td>
<td>4.28</td>
<td>22.64</td>
<td>174.51</td>
<td>1.14</td>
</tr>
<tr>
<td>Pfetch</td>
<td>0.25</td>
<td>0.97</td>
<td>3.88</td>
<td>15.50</td>
<td>67.14</td>
<td>1.72</td>
</tr>
<tr>
<td>Lalign</td>
<td>0.27</td>
<td>1.04</td>
<td>4.20</td>
<td>19.83</td>
<td>93.41</td>
<td>1.44</td>
</tr>
<tr>
<td><strong>G5:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orig</td>
<td>0.12</td>
<td>0.45</td>
<td>1.79</td>
<td>10.30</td>
<td>57.96</td>
<td>0.57</td>
</tr>
<tr>
<td>Fsafe</td>
<td>0.12</td>
<td>0.45</td>
<td>1.79</td>
<td>10.29</td>
<td>54.75</td>
<td>0.59</td>
</tr>
<tr>
<td>Orig+V</td>
<td>0.08</td>
<td>0.24</td>
<td>0.91</td>
<td>4.13</td>
<td>50.49</td>
<td>baseline</td>
</tr>
<tr>
<td>Fsafe+V</td>
<td>0.10</td>
<td>0.38</td>
<td>1.31</td>
<td>5.46</td>
<td>75.85</td>
<td>0.72</td>
</tr>
<tr>
<td>Pfetch</td>
<td>0.07</td>
<td>0.23</td>
<td>0.88</td>
<td>3.65</td>
<td>14.54</td>
<td>1.38</td>
</tr>
<tr>
<td>Lalign</td>
<td>0.06</td>
<td>0.32</td>
<td>1.10</td>
<td>4.45</td>
<td>17.81</td>
<td>1.15</td>
</tr>
</tbody>
</table>

Table 3.6: Running time of 3point on Apple PowerPC G4 and G5

Similar to the SSE library for Intel Pentium 4, there is an Altivec C/C++ interface for PowerPC. Thus we can again rewrite the Fortran code in C with the provided interface. On G5 with VAST/Altivec installed, we can compare our version with the code automatically vectorized by VAST/Altivec. However, on the G4 machine without VAST/Altivec, we need to vectorize the cFsafe by ourselves and use it as the baseline for the performance comparison. On both machines, GNU C/C++ version 3.3 is used and the optimization level is set to O3 for all compilers. The results are shown in Table 3.7 and Table 3.8.

For both 3point and 5point, we can clearly see the performance advantage of Pfetch and Lalign strategies over those of Fsafe and the Fortran version automatically vectorized by VAST/Altivec (Orig+V). Like on the Pentium 4, here we observe similar improvements by using data alignment, vectorized scalar replacement and array padding on PowerPC architectures. As shown in Figure 3.9, for 3point on G5, the speedup becomes larger when the data size increases. We expect the similar behavior for 5point. For Pfetch and Lalign, their performances compare fairly to each other.

Another factor that we observe on PowerPC related to performance is instruction
<table>
<thead>
<tr>
<th>Size(n)</th>
<th>200</th>
<th>800</th>
<th>3200</th>
<th>12800</th>
<th>51200</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>G4:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cf safe</td>
<td>0.60</td>
<td>2.41</td>
<td>9.69</td>
<td>46.46</td>
<td>287.78</td>
<td>0.38</td>
</tr>
<tr>
<td>cf safe+V</td>
<td>0.26</td>
<td>1.01</td>
<td>4.02</td>
<td>23.37</td>
<td>204.70</td>
<td>0.78</td>
</tr>
<tr>
<td>cf safe+VA</td>
<td>0.22</td>
<td>0.86</td>
<td>3.43</td>
<td>23.25</td>
<td>203.84</td>
<td>0.86</td>
</tr>
<tr>
<td>cf safe+VAR</td>
<td>0.21</td>
<td>0.78</td>
<td>3.06</td>
<td>18.71</td>
<td>190.48</td>
<td>baseline</td>
</tr>
<tr>
<td>cf safe+VARS</td>
<td>0.19</td>
<td>0.70</td>
<td>2.72</td>
<td>16.40</td>
<td>190.18</td>
<td>1.09</td>
</tr>
<tr>
<td>cfPfetch</td>
<td>0.36</td>
<td>1.42</td>
<td>5.71</td>
<td>24.88</td>
<td>113.51</td>
<td>0.75</td>
</tr>
<tr>
<td>cfPfetch+V</td>
<td>0.19</td>
<td>0.67</td>
<td>2.65</td>
<td>12.42</td>
<td>63.20</td>
<td>1.50</td>
</tr>
<tr>
<td>cfPfetch+VA</td>
<td>0.17</td>
<td>0.62</td>
<td>2.38</td>
<td>13.42</td>
<td>61.04</td>
<td>1.56</td>
</tr>
<tr>
<td>cfPfetch+VAR</td>
<td>0.18</td>
<td>0.62</td>
<td>2.49</td>
<td>11.98</td>
<td>59.11</td>
<td>1.60</td>
</tr>
<tr>
<td>cfPfetch+VARS</td>
<td>0.14</td>
<td>0.47</td>
<td>1.87</td>
<td>9.13</td>
<td>53.01</td>
<td>2.02</td>
</tr>
<tr>
<td>cLalign</td>
<td>0.36</td>
<td>1.41</td>
<td>5.69</td>
<td>24.90</td>
<td>108.28</td>
<td>0.76</td>
</tr>
<tr>
<td>cLalign+V</td>
<td>0.24</td>
<td>0.93</td>
<td>3.62</td>
<td>18.94</td>
<td>81.38</td>
<td>1.09</td>
</tr>
<tr>
<td>cLalign+VA</td>
<td>0.19</td>
<td>0.67</td>
<td>2.68</td>
<td>12.60</td>
<td>57.03</td>
<td>1.52</td>
</tr>
<tr>
<td>cLalign+VAR</td>
<td>0.19</td>
<td>0.68</td>
<td>2.68</td>
<td>11.63</td>
<td>60.77</td>
<td>1.52</td>
</tr>
<tr>
<td>cLalign+VARS</td>
<td>0.16</td>
<td>0.56</td>
<td>2.20</td>
<td>9.23</td>
<td>41.08</td>
<td>1.96</td>
</tr>
<tr>
<td><strong>G5:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orig+V</td>
<td>0.08</td>
<td>0.24</td>
<td>0.91</td>
<td>4.13</td>
<td>50.49</td>
<td>baseline</td>
</tr>
<tr>
<td>cf safe</td>
<td>0.24</td>
<td>0.93</td>
<td>3.74</td>
<td>15.20</td>
<td>85.40</td>
<td>0.31</td>
</tr>
<tr>
<td>cf safe+V</td>
<td>0.13</td>
<td>0.44</td>
<td>1.72</td>
<td>7.09</td>
<td>65.74</td>
<td>0.59</td>
</tr>
<tr>
<td>cf safe+VA</td>
<td>0.13</td>
<td>0.39</td>
<td>1.46</td>
<td>5.86</td>
<td>70.26</td>
<td>0.62</td>
</tr>
<tr>
<td>cf safe+VAR</td>
<td>0.11</td>
<td>0.35</td>
<td>1.35</td>
<td>5.42</td>
<td>63.30</td>
<td>0.73</td>
</tr>
<tr>
<td>cf safe+VARS</td>
<td>0.10</td>
<td>0.33</td>
<td>1.23</td>
<td>4.93</td>
<td>72.21</td>
<td>0.72</td>
</tr>
<tr>
<td>cfPfetch</td>
<td>0.14</td>
<td>0.53</td>
<td>2.05</td>
<td>8.24</td>
<td>40.95</td>
<td>0.58</td>
</tr>
<tr>
<td>cfPfetch+V</td>
<td>0.10</td>
<td>0.38</td>
<td>1.49</td>
<td>6.04</td>
<td>25.16</td>
<td>0.82</td>
</tr>
<tr>
<td>cfPfetch+VA</td>
<td>0.10</td>
<td>0.33</td>
<td>1.28</td>
<td>5.18</td>
<td>21.07</td>
<td>0.95</td>
</tr>
<tr>
<td>cfPfetch+VAR</td>
<td>0.08</td>
<td>0.27</td>
<td>1.04</td>
<td>4.25</td>
<td>17.29</td>
<td>1.15</td>
</tr>
<tr>
<td>cfPfetch+VARS</td>
<td>0.06</td>
<td>0.24</td>
<td>0.87</td>
<td>3.53</td>
<td>14.64</td>
<td>1.39</td>
</tr>
<tr>
<td>cLalign</td>
<td>0.12</td>
<td>0.47</td>
<td>1.83</td>
<td>7.33</td>
<td>29.63</td>
<td>0.68</td>
</tr>
<tr>
<td>cLalign+V</td>
<td>0.12</td>
<td>0.43</td>
<td>1.72</td>
<td>6.96</td>
<td>28.25</td>
<td>0.72</td>
</tr>
<tr>
<td>cLalign+VA</td>
<td>0.08</td>
<td>0.29</td>
<td>1.12</td>
<td>4.60</td>
<td>20.50</td>
<td>1.05</td>
</tr>
<tr>
<td>cLalign+VAR</td>
<td>0.08</td>
<td>0.27</td>
<td>1.08</td>
<td>4.40</td>
<td>17.81</td>
<td>1.11</td>
</tr>
<tr>
<td>cLalign+VARS</td>
<td>0.07</td>
<td>0.24</td>
<td>0.92</td>
<td>3.69</td>
<td>14.93</td>
<td>1.32</td>
</tr>
</tbody>
</table>

Table 3.7: Running time of 3point (C implementation) on Apple PowerPC G4 and G5
<table>
<thead>
<tr>
<th>Size(n)</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>G4:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cFsafe</td>
<td>1.35</td>
<td>5.49</td>
<td>22.45</td>
<td>109.86</td>
<td>0.37</td>
</tr>
<tr>
<td>cFsafe+V</td>
<td>0.48</td>
<td>2.28</td>
<td>8.96</td>
<td>54.89</td>
<td>0.90</td>
</tr>
<tr>
<td>cFsafe+VA</td>
<td>0.50</td>
<td>2.17</td>
<td>8.20</td>
<td>55.60</td>
<td>0.92</td>
</tr>
<tr>
<td>cFsafe+VAR</td>
<td>0.47</td>
<td>1.96</td>
<td>7.41</td>
<td>49.56</td>
<td>baseline</td>
</tr>
<tr>
<td>cFsafe+VARS</td>
<td>0.41</td>
<td>1.72</td>
<td>7.02</td>
<td>46.45</td>
<td>1.10</td>
</tr>
<tr>
<td>cPfetch</td>
<td>1.11</td>
<td>4.58</td>
<td>18.82</td>
<td>93.05</td>
<td>0.44</td>
</tr>
<tr>
<td>cPfetch+V</td>
<td>0.37</td>
<td>1.70</td>
<td>7.39</td>
<td>46.50</td>
<td>1.12</td>
</tr>
<tr>
<td>cPfetch+VA</td>
<td>0.55</td>
<td>2.16</td>
<td>8.45</td>
<td>53.41</td>
<td>0.90</td>
</tr>
<tr>
<td>cPfetch+VAR</td>
<td>0.57</td>
<td>2.04</td>
<td>6.93</td>
<td>46.47</td>
<td>1.01</td>
</tr>
<tr>
<td>cPfetch+VPA</td>
<td>0.45</td>
<td>1.74</td>
<td>6.13</td>
<td>43.26</td>
<td>1.15</td>
</tr>
<tr>
<td>cPfetch+VPAR</td>
<td>0.53</td>
<td>1.79</td>
<td>5.74</td>
<td>40.15</td>
<td>1.15</td>
</tr>
<tr>
<td>cPfetch+VPARS</td>
<td>0.45</td>
<td>1.56</td>
<td>4.82</td>
<td>38.16</td>
<td>1.30</td>
</tr>
<tr>
<td>cLalign</td>
<td>0.79</td>
<td>3.18</td>
<td>12.99</td>
<td>74.13</td>
<td>0.61</td>
</tr>
<tr>
<td>cLalign+V</td>
<td>0.63</td>
<td>2.00</td>
<td>8.33</td>
<td>53.61</td>
<td>0.91</td>
</tr>
<tr>
<td>cLalign+VA</td>
<td>0.57</td>
<td>2.06</td>
<td>7.33</td>
<td>49.69</td>
<td>0.98</td>
</tr>
<tr>
<td>cLalign+VAR</td>
<td>0.53</td>
<td>1.84</td>
<td>6.37</td>
<td>43.11</td>
<td>1.08</td>
</tr>
<tr>
<td>cLalign+VPA</td>
<td>0.46</td>
<td>1.67</td>
<td>5.89</td>
<td>43.90</td>
<td>1.17</td>
</tr>
<tr>
<td>cLalign+VPAR</td>
<td>0.49</td>
<td>1.75</td>
<td>6.73</td>
<td>41.31</td>
<td>1.12</td>
</tr>
<tr>
<td>cLalign+VPARS</td>
<td>0.44</td>
<td>1.49</td>
<td>5.41</td>
<td>43.35</td>
<td>1.25</td>
</tr>
<tr>
<td>G5:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orig+V</td>
<td>0.29</td>
<td>1.09</td>
<td>4.00</td>
<td>19.24</td>
<td>baseline</td>
</tr>
<tr>
<td>cFsafe</td>
<td>0.40</td>
<td>1.79</td>
<td>7.13</td>
<td>30.81</td>
<td>0.62</td>
</tr>
<tr>
<td>cFsafe+V</td>
<td>0.25</td>
<td>0.99</td>
<td>4.04</td>
<td>18.84</td>
<td>1.06</td>
</tr>
<tr>
<td>cFsafe+VA</td>
<td>0.24</td>
<td>0.88</td>
<td>3.28</td>
<td>18.35</td>
<td>1.18</td>
</tr>
<tr>
<td>cFsafe+VAR</td>
<td>0.24</td>
<td>0.96</td>
<td>3.51</td>
<td>17.75</td>
<td>1.14</td>
</tr>
<tr>
<td>cFsafe+VARS</td>
<td>0.22</td>
<td>0.78</td>
<td>2.98</td>
<td>17.29</td>
<td>1.31</td>
</tr>
<tr>
<td>cPfetch</td>
<td>0.28</td>
<td>1.12</td>
<td>4.47</td>
<td>20.69</td>
<td>0.95</td>
</tr>
<tr>
<td>cPfetch+V</td>
<td>0.19</td>
<td>0.84</td>
<td>3.37</td>
<td>17.98</td>
<td>1.26</td>
</tr>
<tr>
<td>cPfetch+VA</td>
<td>0.24</td>
<td>0.88</td>
<td>3.30</td>
<td>17.97</td>
<td>1.20</td>
</tr>
<tr>
<td>cPfetch+VAR</td>
<td>0.22</td>
<td>0.85</td>
<td>2.95</td>
<td>17.62</td>
<td>1.28</td>
</tr>
<tr>
<td>cPfetch+VPA</td>
<td>0.20</td>
<td>0.82</td>
<td>2.95</td>
<td>17.48</td>
<td>1.36</td>
</tr>
<tr>
<td>cPfetch+VPAR</td>
<td>0.20</td>
<td>0.78</td>
<td>2.67</td>
<td>17.56</td>
<td>1.46</td>
</tr>
<tr>
<td>cPfetch+VPARS</td>
<td>0.20</td>
<td>0.69</td>
<td>2.12</td>
<td>16.51</td>
<td>1.61</td>
</tr>
<tr>
<td>cLalign</td>
<td>0.28</td>
<td>1.14</td>
<td>4.63</td>
<td>21.05</td>
<td>0.93</td>
</tr>
<tr>
<td>cLalign+V</td>
<td>0.26</td>
<td>0.89</td>
<td>3.51</td>
<td>17.57</td>
<td>1.13</td>
</tr>
<tr>
<td>cLalign+VA</td>
<td>0.23</td>
<td>0.81</td>
<td>2.92</td>
<td>16.21</td>
<td>1.30</td>
</tr>
<tr>
<td>cLalign+VAR</td>
<td>0.20</td>
<td>0.77</td>
<td>2.85</td>
<td>17.07</td>
<td>1.35</td>
</tr>
<tr>
<td>cLalign+VPA</td>
<td>0.20</td>
<td>0.69</td>
<td>2.66</td>
<td>16.47</td>
<td>1.49</td>
</tr>
<tr>
<td>cLalign+VPAR</td>
<td>0.19</td>
<td>0.67</td>
<td>2.52</td>
<td>16.31</td>
<td>1.51</td>
</tr>
<tr>
<td>cLalign+VPARS</td>
<td>0.16</td>
<td>0.58</td>
<td>2.16</td>
<td>15.64</td>
<td>1.72</td>
</tr>
</tbody>
</table>

Table 3.8: Running time of 5point (C implementation) on Apple PowerPC G4 and G5
scheduling. Since our code generation includes a loop unrolling phase at the end to remove register copy operations (see Figure 3.6 for example), there are two or more iterations inside the unrolled loop body. If we hoist the vector load statements up as early as possible while still safe, we observe reduced running time almost universally. This only happens to GNU C/C++ compiler on PowerPC, we do not see any effects for Intel C/C++ compiler on Pentium 4. We do not intend to discuss any further about either the compiler or the hardware about instruction scheduling, since it is out of the scope of this dissertation. For reference, we include the running time of the aforementioned eager vector load scheduling strategy (suffix "+S") in Table 3.7 and Table 3.8.

![Graph showing running time of 3point and 5point on PowerPC G5](image)

Figure 3.9: Running time of 3point and 5point on PowerPC G5

3.6 Summary

In this chapter, we presented an integrated scalarization strategy for short vector processors, which considers scalarization, vectorization, and data alignment together. The goal is to ensure the scalarized code fully utilize the underlying vector infrastructure, while at the same time keep the allocated temporary storage minimized. On
the target machines, one important factor related to performance is data alignment. We showed how to achieve the best data alignment performance for both write and read vector accesses. We also described a version of vectorized scalar replacement, which can be integrated into the data alignment process. Our experiments on modern microprocessors demonstrated the effectiveness of this strategy.

Unlike the C language, there is no Fortran extension for using the underlying vector operations. Though Fortran 90 array syntax provide the natural vector representation, the native compiler fail to recognize user-defined arrays that are intended for vectors. To solve this problem, we developed a procedure-outlining strategy that rewrites Fortran loops in C so that vector data types and vector intrinsics can be used.
Chapter 4

Compiling for CELL Processor

In this chapter, we extend the scalarization strategy to CELL processor—a good example of the current trend that microprocessors are moving into the heterogeneous multi-core direction. A CELL processor has two types of computation cores, SPE cores and PPE cores, both support SIMD/vector data types and operations. Thus our scalarization strategy needs to consider the parallelism existed at two levels of granularity on CELL: multiple threads across PEs at the coarse level, and SIMD/vector operations at the fine level. Another issue we need to consider is data movement associated with the scratch pad memory existed on SPEs. Program running on SPEs can only access data in the local scratch pad memory and all data transfers to and from main memory have to be explicitly handled by DMA requests. As to preserving the semantics of array statements, we need to adjust our temporary allocation reduction strategy accordingly for the requirements of parallelization, vectorization and data movement.

In what follows, Section 4.1 gives the architecture details of CELL processor and the problems that an effective scalarization faces, Section 4.2 reviews related work,
Section 4.3 presents a general dependence-based code generation strategy for CELL processor, Section 4.4 discusses how to adjust our temporary allocation reduction strategy for scalarization, Section 4.5 shows experimental results, and Section 4.6 summarizes this chapter.

4.1 Introduction

Computing platforms with multiple processing elements — so-called "multi-core" chips — have been embraced by many chip makers. As a strategy for increasing per-chip performance, adding more cores is an alternative to scaling up operating frequency, which has become difficult because of issues such as heat dissipation. Both Intel and AMD have released dual-core (homogeneous) microprocessors with either shared or separate secondary cache.

A second emerging trend in system design is the use of heterogeneous computing components, either on or off chip. For example, The CELL processor developed by SONY, Toshiba and IBM has 8+1 (heterogeneous) processing elements. Alternatively, attached processing elements such as GPUs and FPGAs have been utilized within a single computing system to accelerate specialized applications.

One disadvantage of these multi-core approaches is that they transfer the burden of achieving high performance from the hardware to the software system or application developer. Thus, an immediate question for these platforms is: How can developers exploit the power of the parallelism? For instance, these elements can be organized by software into many different parallel computing schemes such as task parallelism and pipelined workflow. This question applies to both developing new applications and porting existing applications. Exploring trade-offs like these makes it very difficult to
develop applications for these chips, particularly if a degree of portability is desired. While advanced parallel programmers can, with great effort, use expert knowledge to exploit the advanced hardware features, ordinary users may be left out in the cold. For such users, the new generation of chips desperately need automated tools and compilers that produce code with acceptable efficiency while hiding the details of the underlying hardware.

In the context of scalarization, the above question becomes how to compile programs written in array syntax for these multi-core processors? In this chapter, we focus on the CELL processor, one of the most promising heterogeneous designs. We present an automatic, dependence-based code generation scheme for this chip. We will explain the scheme in two steps:

- First we show how to automatically generate code for general loop nests based on data dependence information on the CELL processor, which involves automatic parallelization, vectorization, data alignment, data movement and synchronization.

- After that, we show how to integrate the scalarization strategy into the code generation process, in particular, how to ensure that the temporary allocation reduction strategy interacts with parallelization part of the code generation well; i.e., they do not exclude each other.

### 4.1.1 CELL Processor

Figure 4.1 shows the architecture of a CELL processor. A single chip contains one PowerPC Processing Element (PPE) and eight Synergistic Processing Elements (SPE) on the same chip. Each SPE supports Altivec-like vector instructions and has a
256K-byte local store memory (LS), while the PPE is a normal PowerPC core with a
two-level cache. The PPE is responsible for scheduling computation tasks onto SPEs.
Computation on a SPE can only access data in its own LS; data movement between
LS and main memory is explicitly controlled by DMA requests via the Element Interconnect Bus (EIB) and Memory Interface Controller (MIC).

![Architecture of CELL processor](image)

Figure 4.1: Architecture of CELL processor

With architecture features such as heterogeneous multi-core, parallelism at both
course (across PEs) and fine (vector, within each PE) granularities, high data trans-
fer bandwidth (200GB/s on chip and 25.6 GB/s off chip), and explicit local memory
control through DMA, a 3.2 GHz CELL processor can, in theory, achieve a peak
performance of more than 200 GFlops per second for single precision floating point
computations. However, actually obtaining high performance levels requires signifi-
cant programming effort because, in the distributed programming tool system, these
architectural features must be explicitly managed by the programmer. This adds
a significant degree of complexity to programming to conventional uniprocessors or
even simple shared-memory parallel systems.
4.1.2 Code Generation for CELL Processor

Over the years, there have been many parallel programming models developed to reduce the burden on ordinary users. They include directive-guided task parallel models such as OpenMP, data parallel models such as HPF, and programming models with language extensions such as UPC and CoArray Fortran. Recently, the DARPA HPCS program has sponsored the development of a new generation of “high-productivity” parallel programming languages such as IBM-X10, Sun-Fortress and Cray-Chapel. However, among these models, OpenMP is to date the only one supported on CELL, and that is by an IBM research compiler [20] rather than the standard distributed tool set.

In this chapter, we present a dependence-based approach to automatically generating code for CELL from a high-level language. Our system performs parallelization, vectorization and data movement optimization automatically. The input program is a single source sequential program without any parallelism directives. The output is a PPE program with many SPE programs, where parallelism is realized by the PPE thread fork-and-joining the SPE threads, as shown in Figure 4.2.

![Fork-and-join execution model and compilation model on CELL](image)

Figure 4.2: Fork-and-join execution model and compilation model on CELL

This parallelism scheme is similar to the OpenMP approach used by the IBM research compiler, except that it uses fully automatic detection of parallelism in place
of user-entered directives. Using information from dependence analysis, the compiler determines whether a loop nest can be parallelized across PEs and vectorized on each PE. Each such loop nest will be partitioned, using a technique called procedure outlining (the opposite of inlining), into a PPE call stub function and a SPE program, as shown in Figure 4.2. In contrast to the OpenMP approach, our strategy can also handle loops carrying dependences, preserving the correct semantics of the program using barrier and a uni-directional synchronization primitives we developed using the on-chip communication mechanisms.

Our dependence-based approach automatically orchestrates and optimizes data movement between the SPE local stores and main memory. For memory references in the original program, multiple data buffers are created on the SPE side and DMA transfer commands are placed accordingly in the SPE program. This strategy also handles the complications of data movement due to the data alignment constraints. In particular, loop peeling on the PPE side can help improve both data movement performance and vector data alignment.

### 4.1.3 Scalarization on CELL Processor

The temporary allocation reduction in our scalarization strategy remains an important optimization on the CELL processor. The reason is that the allocated temporary array not only induces extra memory accesses and may cause more cache misses, but also requires extra data transfer between the LS memory on SPEs and the main memory.

However, the reduced temporary array may prevent the scalarized loop nest from being parallelized, while code generated by the failsafe method can always be paral-
lized. As a solution, we partition array statements that requires temporary array allocation reduction into three categories, and present parallelization strategies for each of them. In particular, we point out that if the reduced temporary array can fit into the local memory on SPEs, they can become completely privatized to each SPE and requires no data transfers. Our experiments show that the reduced data transfers due to the reduced temporary array deliver significant performance improvement.

4.2 Related Work

Increased processor parallelism on chip has become a trend adopted by almost all major chip makers. The parallelism can be found at both coarse and fine level granularity. On coarse level, more and more cores (PEs) are built onto a single chip; on the fine level, vector instruction sets are included and regularly amended (e.g., SSE4). Research in compiling for parallelism at both levels have had a long history with many languages, tools and programming models. However, up till recently, the OpenMP approach, used in an IBM research compiler [20], was only one implemented to support C language on CELL.

While revising this dissertation, several new approaches are developed for programming CELL. Similar to the OpenMP compiler, CellSs [5] is another single source compiler for C language. With CellSs, users need to specify input and output variables for each specified task. In other development, Sequoia [22] requires CELL programming in a new language, while Charm++ [33] requires the encapsulation of computation tasks using a library (Offload API). Compared to these approaches, we compile Fortran programs and requires no user specified directives. Some of these approaches also assume that partitioned work do not communicate with each other.
We, on the other hand, take advantage of the on-chip communication mechanism to improve the application performance.

The dependence-based approach presented in this chapter is related to nearly two decades of work on loop nest parallelization and vectorization work done for various parallel architectures [2, 3, 4, 6, 18, 36, 21, 20, 62, 41, 42]. The data alignment problem on the CELL is very similar to that on SSE and Altivec [21, 62], except on CELL data alignment also affects data movement. Scalar replacement has been developed to improve data reuse at register level [14] and later extended to vector level [49, 62]. In this chapter, we use reference group partitioning to identify the array references that should share a data buffer.

Multi-buffering analysis and placement is very similar to that of software prefetching [13, 40], which was designed to hide the memory latency. On the other hand, optimizing DMA transfers using multi-buffering is similar to array copying [34, 51, 54, 58], which was proposed to eliminate conflict misses in the cache.

Effectiveness of traditional memory hierarchy optimizations for SPE’s local store, *e.g.*, loop tiling, is demonstrated in the IBM work [20]. Since we have not implemented these optimizations in our compiler, we have not yet achieved same amount of speedup as the IBM work for the same benchmark programs. We are currently investigating the optimizations to improve data reuse in SPE’s LS.

Scalarization with a goal to minimize the allocated temporary array on CELL processor has not been studied before this dissertation. On SMP machines, it was pointed out in [48] that failsafe method should be preferred because neither loops carries dependence and thus can be easily parallelized. While on CELL processor, we present solutions that achieve parallelization and temporary allocation reduction at the same time, which delivers a tremendous performance improvement upon the
failsafe method.

4.3 Dependence-based Code Generation

In this section, we present an algorithm for determining whether a loop nest can be parallelized across PEs and vectorized on each of them. Such a loop nest will be procedure outlined into a program for the SPEs, and replaced by a call to a call stub function in the PPE program that manages fork-and-join operations for SPE threads. The SPE program will contain the vectorized loop nest with automatically generated data movement. Synchronization will be generated and placed in both the SPE and PPE programs if necessary. We refer to this loop nest as cellized and this process as cellization.

4.3.1 Parallelization and Vectorization

To analyze if a loop nest can be cellized, we need to determine if there exists a loop in the loop nest that can be parallelized, and a loop (the innermost loop) that can be vectorized on SPE and PPE. These two candidates can be the same; i.e., the innermost loop can be both parallelized and vectorized, assuming that the loop is longer than the four iterations (two for double precision) that would fit in a single vector operation.

Data Dependences The cellization analysis relies on the data dependence information. Figure 4.3 gives an example of a loop nest and its dependence matrix. In the matrix, the anti-dependences in rows 1 to 3 corresponds to dependences from the first, second and third array references on the right hand side to the array reference on the left hand side. The dependence in the first row is carried by the K loop, the
\[
\begin{align*}
\text{DO } & K = 1, W \\
\text{DO } & J = 1, M \\
\text{DO } & I = 1, N \\
A(I, J, K) &= A(I, J, K+1) + A(I+1, J, K) + A(I+1, J+1, K) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{bmatrix}
K & J & I \\
<1 & =0 & =0 & \text{anti} \\
=0 & =0 & <1 & \text{anti} \\
=0 & <1 & <1 & \text{anti} \\
<1 & =0 & >-1 & \text{input} \\
<1 & >-1 & >-1 & \text{input} \\
=0 & <1 & =0 & \text{input}
\end{bmatrix}
\]

Figure 4.3: Example of a loop nest and its dependences

Dependence in the second row is carried by the I loop, and the one in the third row is carried by the J loop. Rows 4 to 6 correspond to the input dependences among the right hand side array references themselves. Note that for the purpose of parallelization and vectorization analysis, input dependences are omitted from consideration.

Cellization Analysis Our cellization analysis algorithm is modified from the Allen-Kennedy loop nest vectorization algorithm [4]. As shown in Figure 4.4, our algorithm searches from the outermost loop towards the innermost loop for a loop that can be parallelized in the remaining dependence matrix. At each step, if no loop is found, the current outermost loop is made sequential and all dependences carried by it are removed from the dependence matrix and the search process is repeated. After a parallel loop is identified, the loop nest is cellizable if the innermost loop is vectorizable on PPE and SPE. Obviously, the algorithm can be relaxed to accommodate loop nests that can be parallelized but not vectorized.

We allow the innermost loop carrying anti-dependences to be parallelized and vectorized so long as it doesn’t have a dependence cycle among different statements. As an exception, an anti-dependence cycle on a statement itself is allowed. To preserve the semantics of a loop that has anti-dependences, we use use a post-store strategy
procedure IsCellizable(LN)
// LN: loop nest \{L1, L2, ..., Ln\} from outermost to innermost
obtain dependence matrix DM for LN
initialize ParallelFlags[1:n] to UNMARKED
while there are loops left in LN
  Search for Li in DM such that all entries in the Li column are "=
  if Li does not exist
    if there are no loops left in LN
      ParallelFlags[n] := PARAVEC
      return IsShortVectorizable(Ln, DM)
    else
      ParallelFlags[current outermost] := SEQUENTIAL
      remove dependences carried by the current outermost loop from DM
      remove the current outermost loop from LN
    else
      ParallelFlags[i] := PARALLEL
      return IsShortVectorizable(Ln, DM)
procedure IsShortVectorizable(L, DM)
// L: the innermost loop
// DM: the dependence matrix after a parallel loop is selected
// or all outer loops are made sequential
for all dependences carried by an outer loop Lk in L1 .. Ln-1 in DM
  remove such dependences from DM
if L carries a dependence cycle among statements
  except an anti-dependence on a statement itself
  return false
for all array references A in L
  if A is not loop invariant to L and not contiguous in memory
    return false
return true

Figure 4.4: Cellization analysis algorithm
using a uni-directional synchronization. As illustrated in Figure 4.5, \( P1 \) needs to temporarily hold the computation results in the buffer only for those writes whose main memory locations are also read by \( P0 \). The buffer for temporarily storing those results on \( P1 \) can completely remain local in LS and not appear in main memory.

![Diagram](image)

Figure 4.5: Parallelizing a loop with anti-dependence

For the example loop nest shown in Figure 4.3, following the algorithm, after the first iteration of the search process, the K loop will be marked SEQUENTIAL and the dependence in row 1 will be eliminated from the dependence matrix; after the second iteration in the search process, the J loop will be marked SEQUENTIAL and the dependence in row 3 will be eliminated; after the third iteration in the search process, the I loop will be marked PARAVEC and sent to check for vectorization. Since the I loop carries an anti-dependence on the same statement and all array references are memory contiguous to the loop, the I loop is vectorizable. Hence the loop nest is cellizable.

**Parallel Code Generation** After a loop nest is identified as cellizable, it will be procedure outlined into a SPE program and replaced with a call to a PPE call stub function which manages SPE task allocation and fork-and-joining of SPE threads. To partition \( n \) iterations of the chosen parallel loop among \( p \) PEs, each PE can get \( \left( \frac{n}{p} + \frac{n \% p}{p} \geq id?1 : 0 \right) \) iterations, where \( id \) is the rank of the current PE. Other than partitioning, parallel code generation also needs to place synchronization accordingly, as shown in Figure 4.6.
procedure Cellize(LN)
    // LN: loop nest \{L1, L2, ..., Ln\} from outermost to innermost
    for loop \(Li\) in LN from outermost to innermost
        if ParallelFlags[i] is SEQUENTIAL
            output \(Li\) in the new loop nest
            if ParallelFlags[i+1] is not SEQUENTIAL
                output barrier synchronization
        else if ParallelFlags[i] is UNMARKED
            output \(Li\) in the new loop nest
        else if ParallelFlags[i] is PARALLEL or PARAVEC
            output \(Li\) with partitioned iterations
    
    vectorize \(Ln\)
    if ParallelFlags[n] is PARAVEC and \(Ln\) carries anti-dependence
        apply post-store to the vectorized \(Ln\)
        insert uni-directional synchronization

Figure 4.6: Cellization code generation algorithm

The vector instruction sets on PPE and SPE are of short vector length (16 bytes) and support only contiguous memory accesses. Though certain patterns of strided accesses can be realized by data permutation afterward, we only consider contiguous memory accesses in this dissertation. Since we already addressed loop based vectorization for SSE and Altivec instruction sets in Chapter 3, we will not discuss them in details again.

For our example loop nest in Figure 4.3, the generated PPE code, PPE stub function code and the SPE code are shown in Figure 4.7. Note that the first element of the computed buffer is stored back after unidirectional synchronization in order to preserve the correct semantics of anti-dependences. The number of elements that need to be post-stored should be no less than the maximal distance of the anti-dependences.

Load Balancing Since PPE forks and joins SPE threads sequentially, a load balancing strategy should consider the cost of thread forking. Assume the cost is
PPEStub1(...) {  
    put loop invariants into data block;
    for(id=1; id<PES; id++)
        spe_thread_fork(&block);
    call PPEStub1(...)
    for k = 1, w {
        for j = 1, m {
            barrier_synch;
            compute my_lb, my_ub;
            for i = my_lb, my_ub
                a(...) = a(...) + ...;
            uni-directional_synch;
        }
    }
}  

SPEmain(...) {  
    dma_get(data block)
    for k = 1, w {
        for j = 1, m {
            barrier_synch;
            compute my_lb, my_ub
            dma_get(b2, b3, b4);
            for i = my_lb, my_ub
                b1(i) = b2(i) + b3(i) + b4(i);
                bt(1:1) = b1(my_lb:my_ub)
                dma_put(b1(my_lb+1:my_ub));
                uni-directional_synch;
                dma_put(bt(1:1));
        }
    }
}  

(a) PPE program (b) PPE stub (c) SPE program

Figure 4.7: Example of code generation

equal to that of executing $c$ iterations of the chosen parallel loop, to partition $n$
iterations across $p$ SPEs, the $k$th PE will get $w_k = n/p + c(p - 2k + 1)/2$ iterations.
This is derived from $w_{k+1} = w_k - c$ and $\sum_{k=1}^{p} w_k = n$. While $c$ could be determined by
estimating the cost of each iteration with program analysis, it could also be decided
with profiling information. If the measured thread forking time and total running time
on one SPE are $t_f$ and $t$, respectively, assuming the parallel loop is the outermost
loop, then $c = nt_f/(t - t_f)$.

**Synchronization Implementation**  The only communication mechanism on
CELL other than DMA transfers that does not need special privilege is mailbox.
Other mechanisms such as SPE-to-SPE signals require the executable be granted
a certain capability (CAP_SYS_RAWIO). According to the Linux man pages, the
capability granting feature is still being developed in the Linux Kernel as of now.
Therefore, we implemented the barrier and the uni-directional synchronization using
mailbox as shown in Figure 4.8. The number associated with each arrow (mailbox send) indicates the mail sending order.

Figure 4.8: Barrier and uni-directional synchronization using mailboxes

### 4.3.2 Data Movement and Alignment

In this section we discuss how to achieve an efficient data movement in between the SPEs’ local memory and the main memory. To do this we must:

1. hide the latency of data movement by multi-buffering so that the DMA data transfer and computation are partially overlapped, if not fully, and

2. use the available bandwidth effectively; *i.e.*, all data brought in should be used and repeated use of the same data should require the data be transferred fewer times as possible.

The second issue is only briefly addressed in this dissertation. We will focus on multiple buffer allocation, DMA data transfer generation and placement.

Figure 4.9 gives an algorithm for analyzing and allocating multiple data buffers. The reference group partitioning in the algorithm is exactly the one used for scalar replacement [14]. We do not yet have a good buffer size estimation algorithm. The data buffer size depends on a set of predefined blocking sizes for the loop nest. Since the maximum size of a single DMA transfer is 16K bytes, it is desirable to allocate a size close to that limit, assuming that the combined code and data buffer fit into the
procedure BufferAllocation(LN)
    // LN: loop nest {L1, L2, ..., Ln} from outermost to innermost
    perform reference group partitioning for all array references
    for the group generator of each reference group
        determine the innermost loop that the generator is loop variant
        determine if the generator is memory contiguous to that loop
    for each array reference in the loop nest
        if it is a group generator
            strip-mine the innermost variant loop if memory contiguous within it
            allocate multi-buffering buffers
            place DMA prefetch current buffer before the loop
            place DMA prefetch next buffer inside the loop
            place DMA check data ready inside the loop
            place buffer index rotation computation at the end of the loop
            replace array reference with allocated buffer
        else
            replace array reference with buffer allocated by the group generator
            adjust subscript by the dependence distance to the generator
    Figure 4.9: Multi-buffering analysis and allocation algorithm

SPE's 256K-byte local memory. For the generated code in Figure 4.7, following the algorithm, the I loop would be strip-mined and multi-buffering data buffers would be allocated.

Another factor affecting data movement performance on CELL is data alignment. It constrains data movement in three ways: vector offset, naturally aligned boundaries, and cacheline alignment.

Vector Offset DMA transfer requires that the last 4 bits of the source and sink addresses be the same. Therefore, when converting an array reference in the main memory space into a local buffer reference, we need to make sure that they have same vector offsets.

Naturally Aligned Boundaries DMA transfer can transfer 1, 2, 4, 8, 16, 16 \( \times k \) bytes (max 16K Bytes) on naturally aligned boundaries. When the starting and
ending addresses are not properly aligned, one has to issue multiple DMA instructions, as shown in Figure 4.10.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DMA(&A, 4 bytes)
DMA(&B, 8 bytes)
DMA(&D, 16 bytes)
DMA(&H, 8 bytes)
DMA(&J, 4 bytes)

Figure 4.10: DMA transferring unaligned data

For DMA transfers that get data from the main memory to the SPE's local memory, we can always over-fetch, i.e., extend the memory transfer region to naturally aligned boundaries on both ends. This is always safe as long as we are careful when allocating buffers. However, we cannot easily use this strategy for the DMA puts from SPEs to the main memory. In short, we have a false sharing problem. Over-storing would require an atomic operation to get the boundary data from main memory, merge with the local data, and put data back into the main memory on the extended boundaries. Implementing such an atomic operation would incur a significant performance penalty.

**Loop Peeling** To reduce the performance penalty related to the unaligned DMA puts, we apply loop peeling to the innermost loop on the PPE side and only compute partitioned loop iterations as multiples of the SPE vector size. The peeled iterations are computed on PPE, while the starting address for a DMA put from an SPE is always at a vector boundary. Loop peeling on the PPE is always safe if the innermost loop is parallelized, or if a loop is chosen to parallelized only when there are no dependences left in the dependence matrix following the algorithm in Figure 4.4. There is a slight issue due to the different rounding modes for floating point on PPE and SPE. We leave it up to the application developers to decide if this discrepancy
actually matters.

**Cache Line Alignment** When a CELL processor performs DMA transfers, the unit data transfer is actually a cache line; i.e., when transferring data as part of a cache line, the bandwidth of an entire cache line will be consumed and the result is masked. On the other hand, when transferring a large data block that is not aligned on cache line boundaries, two whole cache lines will be transferred on the boundaries even though only a part of each cache line is requested. Loop peeling can also be used to ensure data alignment on the cache line boundaries. The wasted cache line bandwidth is relatively small when actually transferring a large data block. This is another reason why a large tile size should be used whenever possible.

### 4.4 Scalarization and Temporary Array Allocation

In this section, we discuss how the scalarization strategy with temporary allocation reduction interacts with the cellization process. During the scalarization process, when temporary array needs to be allocated to preserve the semantics of the array statements, the code generated by the failsafe method can always be easily parallelized and vectorized. As an example, for array statement

\[
A(2:N+1, 2:N+1) = A(1:N, 1:N) + A(3:N+2, 3:N+2)
\]

Figure 4.11 gives the code generated by the failsafe method. As we can see, each of the two loop nests can be easily cellized for the CELL processor independently because neither of them carries dependences.

However, the allocated full size temporary array by the failsafe method not only induces extra memory accesses and may cause cache misses, but also requires extra
DO J = 1, N
    DO I = 1, N
        T(I, J) = A(I, J) + A(I+2, J+2)
    ENDDO
ENDDO
DO J = 1, N
    DO I = 1, N
        A(I+1, J+1) = T(I, J)
    ENDDO
ENDDO

Figure 4.11: Failsafe method for cellization

data transfer between the SPEs’ local memory and the main memory. Therefore, it is important to integrate the temporary allocation reduction strategy developed in Chapter 2 and Chapter 3 into the cellization process.

Based on the relative positions between the loop that loop alignment algorithm allocates the temporary (temporary allocation loop) and the loop that is chosen to be parallelized (parallelization loop), we classify array statements into three categories and present solutions accordingly. The parallelization loop is determined using the scalarization dependence matrix during the scalarization process: following the scalarization algorithm, from the outermost loop to the innermost loop, after a loop is checked that whether a temporary array needs to be allocated or not, it is also checked whether it can be parallelized.

### 4.4.1 Allocation Loop Equals Parallelization Loop

The temporary allocation loop and the parallelization loop are the same only when it is also the innermost loop. The reason is that for an outer loop, if temporary allocation is needed, then it carries dependences and thus cannot be parallelized. Recall that we parallelize the innermost loop even if it carries anti-dependences in Section 4.3.
The parallelization requires loop alignment strategy to post-store the results on the boundaries so that original semantics is preserved. Depends on the dependence pattern, either a barrier synchronization or a uni-directional synchronization is needed.

As an example, for array statement

\[ A(2:N+1) = A(1:N) + A(3:N+2) \]

our loop alignment algorithm will allocated a temporary of size 2 for a scalar processor, as shown in Figure 4.12.

```plaintext
T1 = A(1) + A(3)
DO I = 1, N-1
   T2 = A(I+1) + A(I+3)
   A(I+1) = T1
   T1 = T2
ENDDO
A(N+1) = T1
```

Figure 4.12: Loop alignment method for cellization

The I loop can be parallelized and vectorized with scalar expansion. When the iterations of the I loop get partitioned into blocks among PPE and SPEs, synchronizations need to be inserted such that the values written into array A in memory on one PE are not read by its previous and succeeding neighbors. The pseudo code generated on SPEs is shown in Figure 4.13.

Note that in the generated code two boundary values of A are not stored until after the barrier synchronization, which indicates that all fetch operations have finished, as illustrated in Figure 4.14. The barrier synchronization can be reduced to a uni-directional synchronization if the post-store is necessary for only one neighboring PE, as shown by the example in Figure 4.5 and Figure 4.7 (c).

An important property of the scalarization for this category is that the allocated temporary can completely reside in the LS memory of each SPE; i.e., the allocated
T1 = A(my_lb) + A(my_lb+2)
DO II = my_lb, my_ub-1, TS
   b2(:) = DMA_GET(A(II+1:II+TS+2))
   b1(2:TS+1) = b2(1:TS) + b2(3:TS+2))
   IF (II .eq. my_lb) THEN
      DMA_PUT(b1(2:TS), A(II+2:II+TS))
   ELSE
      DMA_PUT(b1(:), A(II+1:II+TS))
   ENDIF
   b1(1) = b1(TS+1)
ENDDO
call Barrier_Synch
DMA_PUT(T1, A(my_lb+1))
DMA_PUT(b1(1), A(my_ub+1))

Figure 4.13: Pseudo SPE code for temporary allocation in the innermost loop

Figure 4.14: Parallelizing an array statement with anti-dependence

temporary will not show up in the main memory unless PPE takes part in the computation. This is true because for the innermost loop, we only need to allocate a temporary array of constant size \( O(1) \) to preserve the semantics, as proved in Chapter 2.

### 4.4.2 Allocation Loop Inside Parallelization Loop

When the temporary allocation loop is inside the parallelization loop, the only issue is that the temporary allocated by each PE should be privatized on themselves. No synchronization is needed among PEs inside the parallelization loop.
Depending on the size of the allocated temporary array, it may or may not fit completely inside the LS memory of a SPE. If it fits, similar to the previous category, the temporary array can be allocated in the LS memory completely and does not show up in the main memory. As a consequence, no extra data transfer between LS memory and main memory is needed for the temporary array. If it does not fit in the LS memory, the temporary array will have to be allocated in the main memory and transferred between the LS memory and the main memory.

4.4.3 Allocation Loop Outside Parallelization Loop

To increase the possibility of parallelization after temporary allocation, we favor reduced lazy loop alignment described in Chapter 2 because the transformation will make the current loop carry all the dependences and thus make the loops inside parallelizable.

As an example, for array statement

\[ A(2:N+1, 2:N+1) = A(1:N, 1:N) + A(3:N+2, 3:N+2) \]

Figure 4.15 gives the generated code after applying reduced lazy loop alignment. While loop J has to be executed sequentially, loop I carries no dependences and its iterations can be partitioned among PEs for parallelization. For the temporary array T, it will be partitioned among PEs as well and the partitioned part should become private on each PE accordingly. Similar to the discussion for the previous two categories, privatized T does not need to appear in the main memory if it can fit entirely into the LS memory on a SPE.
DO J = 1, 2
DO I = 1, N
    T(I, J) = A(I, J) + A(I+2, J+2)
ENDDO
ENDDO
DO J = 1, N-2
DO I = 1, N
    A(I+1, J+1) = T(I, 1)
    T(I, 1) = T(I, 2)
    T(I, 2) = A(I, J+2) + A(I+2, J+4)
ENDDO
ENDDO
DO J = N-1, N
DO I = 1, N
    A(I+1, J+1) = T(I, J-N+2)
ENDDO
ENDDO

Figure 4.15: Reduced lazy loop alignment method for cellization

4.5 Experiments

Experimental Setup: A dependence-based code generator for CELL (CELLizer) that includes the parallelization, vectorization, multi-buffering DMA transfer and synchronization algorithms described in Section 4.3 is implemented as an extension of our earlier tool for vectorization and code generation for short vector machines with SSE and AltiVec (see Chapter 3), which in turn was based on the D System compilation infrastructure at Rice.

CELLizer accepts a FORTRAN 90 program as source input, performs cellization, outputs a FORTRAN 90 program which will remain on PPE, a collection of PPE stub functions written in C, and a collection of corresponding SPE programs written in C, as we described in Section 4.1. Each SPE program corresponds to a cellized loop nest that is procedure outlined and rewritten in C.
We use the cross-platform compilers (ppuxlc and spuxlc) in the CELL software development kit version 1.1 to compile the PPE call stub functions and the SPE programs. The remaining PPE program in Fortran is translated to C using the f2c converter [23], and compiled with ppuxlc respectively.

The compiled executable is transferred to a 3.2 GHz Cell Blade running Linux OS for correctness verification and performance measurement. The Cell Blade has 1G bytes memory installed, while the swap disk is turned off. We use the Linux library call `gettimeofday()` to measure the time spent in either the original loop nest, or the PPE stub functions and the SPE threads combined. Each code is run 6 times and the minimum running time is kept as record. The speedup of running the generated code on various number of SPEs over running the original loop nest on PPE alone is reported. Our CELLizer has the following compiler options that can be changed.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-cell &lt;arg&gt;</td>
<td>generate code for cell processor with &lt;arg&gt; SPEs</td>
</tr>
<tr>
<td>-plp</td>
<td>loop peeling on PPE to improve data alignment</td>
</tr>
<tr>
<td>-pwk &lt;arg&gt;</td>
<td>work on PPE (&lt;arg&gt; as a percentage to the work of a single SPE)</td>
</tr>
<tr>
<td>-vsr</td>
<td>do vectorized scalar replacement</td>
</tr>
<tr>
<td>-spls</td>
<td>do software pipelined aligned load/store</td>
</tr>
<tr>
<td>-novec</td>
<td>no vectorization, generate sequential C code</td>
</tr>
</tbody>
</table>

Table 4.1: CELLizer options

**Test Cases** We tested three small loop nests $LN1, LN2, ANTI$, shown in Figure 4.16. Each loop nest has an initialization before it and a checksum computation after it. Note that $ANTI$ carries an anti-dependence which usually requires allocating a temporary array and splitting the original loop into two by failsafe method so that each loop can be parallelized, as shown in $ANTI_{MP}$. Our CELLizer, on the other hand, avoids this temporary array allocation by using post-store and uni-directional
synchronization. We also tested two programs from SPEC benchmark, *swim* and *mgrid*. In all test cases, data types were converted to be single precision floating point.

**LN1:**

\[ B(2:N-1) = (A(1:N-2) + A(3:N) + C(2:N-1)) \times 0.34 \]

**LN2:**

\[ A(2:N-1,2:M-1) = A(2:N-1,2:M-1) \]
\[ + (B(1:N-2,2:M-1) + B(3:N,2:M-1)) \]
\[ + B(2:N-1,1:M-2) + B(2:N-1,3:M) \times 0.25 \]

**ANTI:**

\[ A(2:N-1) = (A(3:N) + B(2:N-1) + C(2:N-1)) \times 0.34 \]

**ANTITMP:**

\[ T(1:N-2) = (A(3:N) + B(2:N-1) + C(2:N-1)) \times 0.34 \]
\[ A(2:N-1) = T(1:N-2) \]

Figure 4.16: Testing codes

The performance results are shown in Figure 4.17, Figure 4.18 and Figure 4.19, respectively. For LN1, both a small problem size \( N = 7555333 \) and a big problem size \( N = 44222111 \) are tested. Similarly, \( M = N = 2955 \) and \( M = N = 7255 \) are tested for LN2, \( N = 7555333 \) and \( N = 33222111 \) are tested for ANTI and ANTITMP. For SPEC benchmark programs, the reference problem sizes are tested: \( 1335 \times 1335 \) for *swim* and \( 128 \times 128 \times 128 \) for *mgrid*.

The default set of optimizations includes parallelization across multiple PEs (-cell <arg>), multi-buffering data movement, vectorization, vectorized scalar replacement (-vsr), software pipelined load/store (-spls), and loop peeling on PPE for data alignment (-plp). In Figure 4.17, “pwk10” (“pwk20”) assigns 10 percent (20 percent, respectively) of a single SPE’s work/iterations to PPE, in an attempt to balance the load across all PEs; while “noplp” turns off the loop peeling on the PPE side.
Figure 4.17: Performance of 1D (LN1) and 2D (LN2) stencils

Figure 4.18: Performance of loop (ANTI) with anti-dependence

Figure 4.19: Performance of *swim* and *mgrid*

**Performance Results** Based on the performance data, we have the following observations:
- Programs run as much as 3 times faster on SPE than on PPE. There might be two reasons for this phenomenon. First, the latency of floating point instructions on SPE (6 cycles) is shorter than that on PPE (10-12 cycles). Second, PPE is also responsible for running the operating system.

- We do not get a linear speedup to the number of SPEs used. In fact, what we observe is a curve: the best speedup occurs when less than 8 SPEs are used for small problem sizes. The reason is that thread fork-and-join operations become higher in cost when more SPEs are used, compared to the computation cost on each SPE. We are still investigating the effectiveness of the load balancing strategy described in Section 4.3.1. Nevertheless, increasing the problem size can mitigate this problem by offering each SPE more work so that thread fork-and-join costs become less obvious, as illustrated in Figure 4.17 and Figure 4.18.

- Our strategy of loop peeling on the PPE side to help improve DMA data alignment works well. Compared to not performing loop peeling, our strategy achieved an average of 9 percent speed improvement for the 1D stencil and 7 percent for the 2D stencil, as shown in Figure 4.17. Note that for the two-dimensional array in 2D stencil, the loop peeling strategy will dynamically calculate the number of “I” iterations it needs to peel within each outer “J” iteration, so that on SPEs the array reference on the left hand side is always properly aligned when entering the “J” loop.

- Comparing the performance between ANTI and ANTITMP in Figure 4.18, we can clearly see the benefits of our parallelization strategy using post-store and uni-directional synchronization to handle the anti-dependences. The extra memory traffic in ANTITMP and extra thread fork-and-joining costs have
caused heavy performance penalty. This result also suggests that loop fusion combined with array contraction work well for the CELL processor if the resulting loop with dependences can be parallelized with post-store strategy and proper synchronization.

- When running whole applications on CELL, we can tune each loop nest independently to determine the number of SPEs for the best speedup, instead of using the same number of SPEs for all of them. Figure 4.19 shows that the tuning result for three most time-consuming loop nests, $L3$, $L5$, and $L8$ in the *swim* benchmark. Unfortunately, they all achieve the best speedup with two SPEs, while the whole application achieve the best speedup with three SPEs. The discrepancy is caused by collecting timing information of $L3$, $L5$ and $L8$ for only 20 iterations, while they are iterated for 1200 iterations in *swim*, as specified by the SPEC benchmark.

- Finally, contrary to what people may think, having PPE work on partitioned workload doesn’t seem to help much due to the relative slowness of the PPE, as shown in Figure 4.17. However, loop peeling on the PPE can help improve DMA data alignment. Load balancing across all PEs needs further investigation.

**Thread Cost Reduction** The performance penalty because of the cost of SPE thread fork-and-join operations is worth noting, since the benefits of using more SPEs are diluted by this cost. Table 4.2 gives the measured time (in micro seconds) of creating a number of SPE threads on our testing machine.

To reduce this cost, one strategy is to load all SPE functions into SPE’s LS only once and have a dispatch function to invoke the proper function upon receiving a number from PPE. At compile time, each cellized loop nest is assigned a unique
Table 4.2: SPE thread creation cost

<table>
<thead>
<tr>
<th># of SPEs</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (μsec)</td>
<td>4259</td>
<td>7914</td>
<td>11428</td>
<td>14964</td>
<td>18915</td>
<td>22836</td>
<td>26642</td>
<td>30324</td>
</tr>
</tbody>
</table>

number to correlate the PPE stub function and the SPE function. At run time, when PPE enters the PPE stub function, it will not create threads to run SPE functions. Instead, it sends the number to the dispatch function running on the SPE and the corresponding SPE function is invoked. We credit this strategy of creating SPE threads only once, to IBM's research compiler group, who implemented it in their OpenMP compiler [20].

We implemented the thread cost reduction strategy in our compiler. Note that the performance of test cases LN1, LN2 and ANTI will not be affected since each only contains one loop nest that is iterated once. For ANTITMP, since there are two loops, performance of the second loop can be improved by not creating threads, as shown in Table 4.3. The performance of the first loop become worse because previously, thread creation and SPE computation are overlapped, now they are separate and the threads are created when the first loop is encountered. These measurements are obtained with the same small problem size N = 7555333 as used in Figure 4.18.

Table 4.3: Running time of ANTITMP before and after thread cost reduction

<table>
<thead>
<tr>
<th># of SPEs</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop 1 (μsec)</td>
<td>728955</td>
<td>513975</td>
<td>366542</td>
<td>340099</td>
<td>427146</td>
<td>456095</td>
<td>465611</td>
<td>371477</td>
</tr>
<tr>
<td>loop 2 (μsec)</td>
<td>82450</td>
<td>50501</td>
<td>35276</td>
<td>42472</td>
<td>40684</td>
<td>38768</td>
<td>40762</td>
<td>42567</td>
</tr>
<tr>
<td>After:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop 1 (μsec)</td>
<td>913630</td>
<td>659722</td>
<td>703034</td>
<td>794462</td>
<td>666937</td>
<td>581556</td>
<td>584393</td>
<td>579709</td>
</tr>
<tr>
<td>loop 2 (μsec)</td>
<td>89426</td>
<td>47565</td>
<td>33543</td>
<td>27284</td>
<td>23668</td>
<td>22079</td>
<td>22081</td>
<td>21140</td>
</tr>
</tbody>
</table>

Since the cellized loops in swim and mgrid benchmarks are iterated many times,
the thread cost reduction can help even more. Figure 4.20 clearly demonstrates the performance improvement after implementing the strategy. The experimental setup are exactly the same as that used for Figure 4.19. The performance gain of \textit{mgrid} is not comparable to that of \textit{swim} because our current implementation does not cellize the loop nests that interpolate and extrapolate between the fine and coarse grids, since they access non-contiguous memory. However, the strided memory accesses in an affine pattern can be vectorized effectively, as studied in [42]. Implementing the strategy to cellize such loops is one of our ongoing efforts.

![Graph](image.png)

Figure 4.20: Performance for \textit{swim} and \textit{mgrid} before and after thread cost reduction

Though extremely helpful, this thread cost reduction strategy is not scalable because the 256KB LS memory on a SPE may not be able to accommodate all SPE functions at once. A code partitioning and overlay scheme described in [20] should be exposed and included in CELL SDK for dynamically loading and executing SPE functions from the SPE side. After that, function code can be prefetched or cached just like data. We leave this issue to IBM.
4.6 Summary

In this chapter, we presented a dependence-based automatic code generation strategy for the CELL processor. In contrast to approaches that use explicit parallelism directives, such as OpenMP, it doesn’t require that the user perform dependence analysis manually and determine the legality of the hand-specified parallelization. Besides automatic parallelization, vectorization, and multi-buffering DMA data transfer generation, our CELLizer can also insert two kinds of synchronization automatically to preserve the correctness of a program.

Extending our scalarization strategy that reduces the temporary array allocation to the CELL processor, we showed that the scalarized code can still be parallelized. In particular, the reduced temporary array requires less data transfers between the LS memory of SPEs and the main memory, which can be completely eliminated if the temporary array fits entirely into the LS memory and becomes privatized.

Our performance study verified the correctness of our code generation strategy and the usefulness of performing loop peeling on the PPE side to help improve data alignment. We also demonstrated the great benefits of reducing the size of the allocated temporary array in our scalarization strategy compared to the failsafe method. After implementing a strategy to create threads and load all functions at once, the performance decrease because of the cost of thread fork-and-joining operations are removed. We are currently investigating methods to cellize more loop nests in applications.

To better utilize the available bandwidth to chip, we must improve the amount of computation per transferred data element. One way to do this is through a more thorough use of loop fusion and tiling. This is a goal of our ongoing research.
Chapter 5

Performance Tuning of Loop Fusion

5.1 Introduction

After the scalarization process, array statements are converted into loop nests. It is often possible to apply loop fusion on these loops to further improve the memory hierarchy performance by bringing the data reuse closer in the fused loops. This is the responsibility of a compiler since loop fusion cannot be naturally expressed using array syntax.

The difficulty of loop fusion is to statically select proper sets of loops to fuse, given that there are many different fusion configurations to choose from. Though many vendor compilers have implemented loop fusion as a standard optimization to improve memory hierarchy performance, they employ heuristics and performance models to choose the fusion configuration instead of examining all possible ones, because of the compilation time constraints.
Recent research has resorted to empirical performance tuning to determine the best configuration of optimizations on a specific architecture [54, 32, 25, 15]. However, most of these research focus on optimizations that are easier to parameterize such as loop blocking and loop unrolling, and none of them has formalized the parameter space of loop fusion and explore the properties of the loop fusion space. The related loop fusion work in [46] only searches through the space of pair-wise loop fusion rather than the entire space of reordering loop fusion.

In this chapter, we propose a strategy to parameterize loop fusion such that the entire configuration space of reordering loop fusion can be represented by a set of integer parameters. The parameterization process produces a single parameterized version of the original program, outputs the dependence graph and the other analysis information such as register pressure estimate. Our empirical tuning engine can then generate and evaluate all legal configurations and identify the best one. However, the space of loop fusion configurations is usually very large and hence it is very expensive (if not infeasible) to evaluate the whole space to find the optimal configuration. In order to obtain insights into how to design effective searching techniques and search space pruning heuristics, we conduct a study of the properties of the loop fusion space.

In what follows, Section 5.2 reviews related work, Section 5.3 explains our parameterization and tuning strategy for loop fusion, Section 5.4 presents our experimental results, and Section 5.5 summarizes this chapter.
5.2 Related Work

Like other loop transformation techniques (e.g., blocking, distribution, interchange, skewing, and index-set splitting [4]), loop fusion can significantly improve the memory hierarchy performance of scientific applications. However, optimal fusion is NP-complete in general [19]. Kennedy and McKinley [31] proposed a typed fusion heuristic, which can achieve maximal fusion for loops of a particular type in linear time. Kennedy [30] proposed a weighted fusion algorithm that aggressively fuses loops connected by the most heavy edges (edges that signal the most data reuse between loops).

Recently empirical tuning has attracted attention from research community due to the fact that complex machine architectures pose significant challenges to compilers in estimating the runtime behavior of applications. The related work in empirical tuning of applications can be roughly divided into two categories: domain-specific approaches which use specialized kernel-specific code generators and search engines that exploit problem-specific knowledge and representations, and compiler-based approaches which extend traditional compiler infrastructures with parameterized transformations and search engines.

ATLAS [54] and FFTW [25] are two examples of the domain-specific approaches. During the installation (tuning) phase on a specific architecture, ATLAS generates many versions of the matrix multiplication code that explores the space of optimizations and their parameters related to many architecture characteristics, such as the loop blocking size, array copy, the number of registers, the instruction cache size, and the availability of multiply and add instruction. These generated code will be run on the target machine and the one with the best performance is determined empirically. This approach is often categorized as cache conscious. While categorized as a cache
oblivious approach, FFTW takes advantage of the recursive nature of the algorithm and stops the recursion when the problem size can be efficiently computed with the available machine resources. Lately, Yotov et al [60] pointed out that by considering all parameters that ATLAS explores, a static performance model can be used to determine the best code that is performance-wise comparable to the one determined by ATLAS empirically. However, how to automatically generate such a model for general applications remains a question.

Compiler-based tuning approaches belong broadly to the area of feedback-directed optimization [50], which includes superoptimizers, profile-guided and iterative compilation, dynamic optimization, and empirical tuning of the compiler itself. Earlier work on iterative compilation regarding loop nests, of which this work is an instance, has studied the tuning of loop blocking sizes and unrolling factors [32, 44, 47, 15] and more recently, the pair-wise loop fusion [46]. Key common problems are finding a suitable parameterization of the transformations and exploring effective searching heuristics and search space pruning heuristics.

5.3 Empirical Tuning of Loop Fusion

This section presents our techniques for empirically applying loop fusion to optimize scientific kernels. Figure 5.1 shows our overall framework, which includes the following three components.

- **Optimizer** A source-to-source compiler that reads in an application, performs dependence analysis to determine opportunities of applying loop fusion, then generates two outputs: a parameterized version of the transformed code, and the configuration space of the transformation. A template of the parameterized
code will be shown shortly in sub-Section 5.3.1. The configuration space includes a sequence of integer variables used in the parameterization, and a collection of constraints on their values.

- **Search Engine** An empirical configuration generator that chooses integer values for the loop fusion parameters based on two inputs: the configuration space generated by the optimizer and the collected performance measurements of applications.

- **Code Generator** A simple configuration applicator that takes the parameterized code generated by the optimizer and the configuration of parameter values by the search engine, instantiates the parameters with their corresponding values, thus generates a clean code with no parameters. The clean code can subsequently be compiled by the vendor compiler to produce an executable of the transformed code. The performance of the executable is measured on the target machine and the results of measurements are recorded and used by the search engine to generate the next configuration, until a satisfactory configuration is found.

In our framework, we apply the optimizer only once to generate the parameterized code and the configuration space. The result of entire compiler analysis is encoded within the parameterization. Consequently, the search engine and the code generator do not need to perform any additional dependence analysis. Our iterative empirical tuning process thus avoids recompiling the application every time a new version is generated.
5.3.1 Parameterization of Loop Fusion

Given as input a sequence of loops, as shown in Figure 5.2(a), our parameterization of loop fusion transformations is shown in Figure 5.2(c). The parameterization is based on the observation that each loop fusion transformation partitions the input loops \( l_1, l_2, ..., l_n \) into a sequence of groups, \( G_1, G_2, ..., G_m \), where \( 1 \leq m \leq n \), so that all the statements in each group are fused into a single loop. Figure 5.2(b) shows a template of the transformed code after applying a conventional loop fusion, where \( G_i \) (\( i = 1, ..., m \)) represents the bodies of all the loops that belong to the fusion group \( g_i \). Each clustering of the original loops thus uniquely determines a single loop fusion transformation.

In order to parameterize the results of applying arbitrary loop fusion transformations, we need to explicitly model the relation between the clustered groups of a fusion transformation (\( G_1, ..., G_m \) in Figure 5.2(b)) and the collection of statements in the original code (\( S_1, ..., S_n \) in Figure 5.2(a)). In Figure 5.2(c), we use a sequence
of integer variables, $A_1, A_2, \ldots, A_n$, to model this relation. Specifically, $\forall i = 1, \ldots, n$, if $A_i = j$, then the original statement $S_i$ in Figure 5.2(a) belongs to the clustered group $G_j$ in (b). The values of $A_1, \ldots, A_n$ therefore comprise the configuration space of different fusion transformations.

As shown in Figure 5.2(c), our parameterized code has an outermost loop $l_j$, which enumerates the clustered groups of an arbitrary loop fusion transformation. Each iteration of loop $l_j$ enumerates the code of the clustered group $G_j$ in Figure 5.2(b). Specifically, the fused loop of group $G_j$ is loop $l_i$ in Figure 5.2(c), and the body of the fused loop includes every statement $S_k$ ($k = 1, \ldots, n$) such that $A_k === j$; i.e., statement $S_k$ is assigned to the fusion group $G_j$ by the current fusion configuration.
The code in Figure 5.2(c) contains \( n + 1 \) parameters: \( m \), the number of clustered groups by loop fusion, and \( A_k \) \((k = 1, 2, ..., n)\), the index of the fusion group that statement \( S_k \) belongs. When these parameters are instantiated with different integer values, the code in (c) is equivalent to the conventional output of different loop fusion transformations, which we call clean code. For instance, if \( m = A_1 = A_2 = ... = A_n = 1 \), the code in Figure 5.2(c) is equivalent to fusing all the loops in Figure 5.2(a) together. If \( m = n \), and \( A_k = k \) \( \forall k = 1, ..., n \), the code in (c) is equivalent to the code in (a), where none of the loops are fused. If \( m = n/3 \), and \( A_k = k/3 \) \( \forall k = 1, ..., n \), the code in (c) is equivalent to fusing every three loops in the original code in (a).

Because different instantiations of the fusion parameters can represent the entire configuration space of partitioning individual loops into different groups, the parameterized code in Figure 5.2(c) can be used to represent outputs of arbitrary fusion transformations on the loops in (a). Instead of applying heuristics to determine how to optimize application performance through loop fusion, we have implemented our loop fusion transformation to generate the parameterized output in (c) instead. To guarantee that only valid transformations are applied, our compiler additionally outputs a collection of constraints on the values that can be assigned to the parameters.

### 5.3.2 Configuration Space

As shown in Figure 5.1, after our optimizer generates the parameterized code and the configuration space of applying loop fusion transformations, a search engine exploits the configuration space of the parameters and tries to find a configuration that provides the best performance.

Assume that the loops in Figure 5.2(a) can be fused in arbitrary orders, the
number of configurations that fuse $n$ statements into $m$ loops can be modeled using the following recursive formula: $F(n, m) = F(n, m - 1) + m \times F(n - 1, m)$, where $F(i, i) = F(i, 1) = 1$. This is exactly the Sterling number of the second kind [27], the number of ways to partition $n$ objects into $m$ nonempty subsets. The total number of fusion configurations is therefore $\Sigma_{m=1}^{n} F(n, m)$, and the lower and upper bounds are $n!$ and $F(n, 2) = 2^{n-1} - 1$ respectively. The entire configuration space of loop fusion for $n$ loops is therefore at least exponential.

In real applications, however, the number of valid configurations is much smaller because of the dependences between statements. Suppose that the dependence constraints between statements in a sequence of $n$ loops require that all the loops must be fused in exactly the order they appear in the original code; that is, if $S_i$ and $S_j$ ($i \leq j$) in Figure 5.2(a) are fused together, then all loops in between $\{S_k| i < k < j\}$ are also fused with $S_i$ and $S_j$. In this case, the number of configurations that fuse $n$ loops into $m$ loops is $\binom{n-1}{m-1}$ and hence the total number of fusion configurations is $2^{n-1}$. Figure 5.3 lists the number of different configurations for both reordering fusion transformations (statements in Figure 5.2(a) can be arbitrarily reordered) and pair-wise fusion transformations (no reordering is allowed between statements in (a)).

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reordering</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>15</td>
<td>52</td>
<td>203</td>
<td>877</td>
<td>4140</td>
<td>21147</td>
<td>115975</td>
</tr>
<tr>
<td>Pair-wise</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
</tbody>
</table>

Figure 5.3: Number of fusion configurations

dence information associated with these loops, our search engine will only generate the legal reordering fusion configurations. As an example, for a list of eight loops we
studied in the Riemann code (see Section 5.4), the total number of legal reordering fusion configurations is 982.

The fusion space discussed above is only related to one-level fusion. Now we extend the discussion to multi-level fusion, in which case $S_1, S_2, \cdots, S_n$ can be fused together from level 1 (outermost) to level $d$ (innermost). The fusion space is larger because it is possible that $S_i$ and $S_j$ are fused at level $k$, but not at level $k + 1$. In this case, the total number of fusion configurations for pair-wise fusion is $(d + 1)^{n-1}$, while that for reordering fusion is very difficult to derive.

5.3.3 Code Generation

As shown in Figure 5.1, after the search engine determines a configuration to perform loop fusion transformation, the code generator translates the parameterized code in Figure 5.2(c) into the equivalent output of a conventional loop fusion (clean code), shown in Figure 5.2(b). The clean code is then compiled into executable by invoking the vendor compiler on the target machine.

Our code generator is a simple source-to-source translator that requires no dependence analysis. Specifically, given the parameterized code in Figure 5.2(c), it unrolls the outer loop $l_j$, substitutes all parameters with their corresponding values, then removes all conditional branches inside $l_i$ that evaluates to false after parameter substitution. The translated code is identical to the result that would be generated by a conventional loop fusion transformation.
5.3.4 Directly Executing Parameterized Code

Our parameterized code in Figure 5.2(c) can be compiled into executable even without translation. This feature allows us to dynamically measure the performance of different fusion configurations at runtime. Specifically, we can invoke the vendor compiler to generate a single executable from the parameterized code in Figure 5.2(c), and then use the search engine to set the values of the parameters in (c) at runtime. As directly executing the parameterized code avoids generating and compiling a different clean code for each fusion configuration, more flexibility may be gained.

We have built our framework so that loop fusion configurations can be tuned both at runtime, where the parameterized code are directly compiled and profiled, and at installation time, where the code generator is first applied to translate the parameterized code into clean code, and the performance of the clean code is then measured. An interesting question is whether we can use dynamic loop fusion to identify a small subset of whole configuration space with good performance, followed by performing static loop fusion only on the subset to speed up the search process.

5.4 Experiments

We have shown in Section 5.3.2 that the number of configurations in the space of reordering loop fusion is exponential. To determine the best configuration by exhaustively generating and evaluating all configurations would be very time consuming, if not infeasible. Therefore, we need to find strategies to (1) speedup the tuning process itself, (2) search heuristically, and (3) prune the search space. For the first goal, in our tuning framework, we can use dynamic loop fusion to help static loop fusion if it takes less time to evaluate configurations of loop fusion dynamically than to do so
statically, and if the measured performance correlate to each other. For the second and the third goals, we believe that a thorough study of the properties of the loop fusion space of a relatively small number of loops is required.

5.4.1 Experimental Setup

The study is conducted in the tuning framework shown in Figure 5.1, which has been implemented using the loop transformation work developed by Yi and Kennedy [59]. The implementation is now available in both D System (for Fortran) at Rice and in ROSE (for C/C++) at LLNL. We use a simple implementation of the search engine that exhaustively enumerates all configurations.

We applied loop fusion to Riemann, a kernel which is a part of a piecewise parabolic method (PPM) code [16]. PPM is an algorithm for solving hyperbolic equations, typically from simulations of computational fluid dynamics and hypersonic flows, and is part of the benchmark suite used in procurement evaluations for the Advanced Simulation and Computing (ASC) Program within the U.S. Department of Energy. PPM is moderate in size (the 1D kernel consists of about 50 loops when unfused), compute-bound, can be completely fused (at the cost of excessive register pressure), and has been optimized by others using manual and semi-automated techniques that include fusing all loops [57].

We considered fusing just the core relaxation component that dominates the overall running time of Riemann. This subcomputation has 8 loops which can be completely fused, and these loops are iterated 5 times. (The kernel also contains an additional 10 pre-processing loops before relaxation, and 27 post-processing loops.) Focusing on just the 8 loops permits an exhaustive study of the search space which
consists of $2^{8-1} = 128$ pair-wise fusion configurations. These 128 pair-wise configurations roughly cover one eighth of the space of 982 valid reordering fusion configurations.

For the space of pair-wise loop fusion for $n$ loops, we can encode the fusion configurations with $n - 1$ bits, while the $i$-th bit indicates whether loop $i$ is fused with loop $i + 1$ (bit value 1 for fused, 0 for not fused). For 8 loops, we use bit strings of length 7, $b_1 b_2 \cdots b_7$, where $b_i = 1$ if and only if loops $i$ and $i + 1$ are fused. The bit strings are translated into corresponding integer values of the configuration parameters (shown in Figure 5.2(c)) in our implementation.

Since we use bit strings of length 7 to represent configurations of applying loop fusion to Riemann, the configuration space is a 7-dimensional space. To ease the presentation of raw data we linearize this space (e.g., see Figure 5.4) using a Gray code ordering, where neighboring points in the linearization differ by only 1 bit; i.e., differ in only 1 pair of loops being fused or not fused. The specific Gray ordering we use is a binary-reflected Gray code [7]; we take the first point to be linear index 0 and configuration 0000000 (all unfused), and the last point to be linear index 127 and configuration 1000000. The all-fused case, 1111111, is linear index 85.

We performed experiments on two architectures: a 2.4 GHz Intel Xeon with 8 floating point registers, 8KB L1 cache, 512KB L2 cache; and a 650 MHz Sun UltraSPARC IIe with 32 registers, 16KB L1 cache, 512KB L2 cache. We use the Intel C/C++ compiler 9.0 with "-O3" on the Xeon, and the Sun Workshop 9.0 C/C++ compiler with option "-O5" on the UltraSPARC.

For each configuration we observed hardware counters using PAPI [12], recording process-specific cycles, instructions, cache and TLB misses, and branch mispredictions. To reduce the sensitivity of our reported counts to system noise, we measure
each metric for each fusion configuration 9 times and report the minimum. The time to run the kernel is roughly a few tenths of a second on the Xeon and a few seconds on the UltraSPARC.

5.4.2 Parameterized vs. Conventional Output

There is a potential time and accuracy trade-off in choosing to use either parameterized or conventional output. Empirical tuning using the parameterized output is more flexible in that we need to produce only a single executable which is then dynamically adapted for each configuration at runtime, whereas using conventional output needs to repeatedly invoke both the code generator in Figure 5.1 and the vendor compiler to generate executable for each configuration. However, the parameterized code has a much more complex control structure due to the configuration logic. If we choose to use the parameterized form to search for best configuration, we must ask how accurate this search will be.

We compare the accuracy of using parameterized and conventional output in Figure 5.4, which shows the running time (in cycles) of parameterized output and conventional output for the 128 fusion configurations on the Xeon platform. Observe that the two types of implementations qualitatively track one another. However, the parameterized codes take up to 25% more time to execute due to their complex control structure. We can further quantify the similarity of the two curves using statistical correlation, where a correlation close to 1 indicates a strong increasing linear relationship, -1 indicates a strong decreasing linear relationship, and 0 indicates no linear relationship. On the Xeon, this correlation is 0.85, while on the UltraSPARC it is 0.75, both indicating moderate-to-strong linear relationships. However, these
relationships are not perfect and so there may be performance estimation inaccuracy when using parameterized output.

Figure 5.5 shows the extent to which the parameterized codes reproduce other features of the execution behavior (like cache misses) of the conventional codes on the Xeon. Specifically, we show the ratio of measurements for the parameterized codes to those of the conventional codes. Cycles, store instructions, and L2 and TLB misses—expected to be the most expensive misses—are well-reproduced, showing errors of 25% or less. However, there are uniformly many more load instructions, and in several cases, many more L1 misses. The excess loads can be explained by two characteristics of the parameterized output: (1) the need for additional integer variables to describe the configuration, and (2) an inability of the vendor compiler to apply scalar replacement to reduce memory accesses. The spikes in L1 misses tend to occur in configurations that introduce a considerable degree of fusion. Nevertheless, we expect L1 misses to be relatively cheaper than L2 and TLB misses, so L1 miss behavior may be less necessary to reproduce accurately. Results on the UltraSPARC (not shown) are qualitatively similar except that the largest ratios are smaller (less than 1.7).

We also report on the turnaround time to evaluate each point in the search space. On the Xeon, we observed an average time of 1.89 seconds to generate and compile a conventional output, and 0.18 seconds to execute it, compared to an average of 0.67 seconds just to execute the parameterized version. The overhead of the parameterized output is due to extra control logic, the function call to the runtime configuration generator, and the missed optimization opportunities from the vendor compiler. On the UltraSPARC, it takes 3.0 seconds to generate and compile the conventional output, and 1.3 seconds to execute, while it takes 3.5 seconds for the parameterized version
Figure 5.4: Correlation of cycles on Intel Xeon

Figure 5.5: Ratio of measurements on Intel Xeon: parameterized vs clean version
to execute. Thus, the relative costs will depend on the application itself and the underlying architecture. However, for cases that dynamic evaluation is much faster than the static evaluation and there is a strong correlation between the measurements of these two, we may use the dynamic evaluation to prune the search space first so that static evaluation only searches in the pruned space, and thus speedup the search process.

5.4.3 Properties of the Fusion Configuration Space

We present a preliminary characterization of the fusion configuration space, based on our exhaustive search data, focusing exclusively on the conventional output. We describe several properties of the search space, each implying a possible search space pruning technique.

The distribution of performance in the configuration space can be summarized as follows. On Xeon, the speedup over the completely unfused case is between 0.88 and 1.06, with a standard deviation (σ) equal to .039. There are 90 configurations within 5% of the best speedup and 112 points within 10%. On Sun Sparc, the speedup is between 0.97 and 1.11, with σ = .026. There are 19 points within 5% of the best speedup and 94 points within 10%. Although these data indicate there are many implementations with reasonably good performance, we note that the penalty for making a mistake can be significant: on the Xeon, choosing a poor configuration leads to a “speedup” as low as .88.

The Gray ordering reveals relationships among fusion configurations that could be exploited by a guided search. Figure 5.4 gives the running time in cycles, with all configurations linearized along the x-axis by a binary-reflected left-to-right Gray code
(see Section 5.4.1). There is a qualitative mirror-symmetry in several places, indicating which fused pairs do not affect performance too much. For example, comparing configurations 0–63 to 127–64 indicates that fusing loops 1 and 2 (bit $b_1$) has little effect on the running time. However, segments 0–15 and 31–16, which differ only in bit 3, are not mirror symmetric, suggesting that fusing loops 3 and 4 has a non-trivial effect on performance. In much larger search spaces, it should be possible to detect these interactions quickly by sampling.

Though the curves are not smooth in Figure 5.4, many of the “peaks” and “valleys” are localized (e.g., see the grouped peaks at 20–23, 40–43, 80–87, 104–107). This observation suggests gradient-descent type methods with random perturbations may be an effective search technique.

We consider pruning the search space using a purely static estimation of register pressure. The number of registers is a limited number on a specific architecture. It is a constraint that should be observed by the loop fusion algorithm so that the fused code does not exceed this limit. We estimate the number of registers needed to keep every array reference in a register for the purpose of data reuse as described by Carr and Kennedy for unroll-and-jam [14]. We assign to each configuration the maximum estimated register pressure among all loops after fusion. Of course, many configurations may have the same register pressure estimate.

We show the distributions of speedup as a function of this estimated register pressure using box-plots, as shown in Figure 5.6 for the Xeon (left) and the UltraSPARC (right). At each register pressure value, we show the minimum, maximum, median by short horizontal lines, and the 25th and 75th percentiles of points by trapezoids. On the Xeon, which has only 8 scalar floating point registers, the best speedup is achieved when the register pressure is 8; when the register pressure exceeds 11, the
Figure 5.6: Speedup vs. register pressure on Xeon (left) and UltraSPARC (right) performance drops. By contrast, the UltraSPARC, which has 32 registers, achieves a maximum speedup when there is a high-degree of fusion (in this case, all loops fused). This confirms our intuition that fusing too much can cause performance degradation due to the limited resources such as registers. However, observe also that the register pressure estimate is not enough to identify the optimal configuration since there is generally a wide spread in performance at each register pressure value. In the absence of other possible static properties of the code, this observation suggests that the register pressure estimate could be combined with some form of local search.

5.5 Summary

In this chapter, we studied the empirical tuning of loop fusion. While loop fusion can be applied on the loop nests after scalarization to improve the data reuse across different array statements, we focus on the theory and practice related to tuning the loop fusion transformation for best performance. In theory, we developed a strategy to parameterize the configuration space of loop fusion with a set of integer parameters, to
parameterize the loop fusion transformation so that parameterized code is generated, and to tune loop fusion both statically and dynamically. The benefits of our approach are that the dependence analysis associated with loop fusion is performed only once, and the parameterized code itself can be tuned dynamically at run-time.

However, in practice, the configuration space of loop fusion is huge (at least exponential to the number of loops), which makes running and comparing all configurations very time consuming, if not infeasible. Thus we need solutions to speedup the tuning process. We investigated three strategies by studying an entire search space of 8 loops in a sample program that was originally written in Fortran 90 array code.

- **Dynamic Tuning** In cases that executing parameterized code dynamically for a configuration is faster than executing statically for the same configuration, which requires generating clean code and recompiling with a vendor compiler, if there is a strong correlation in between the performances obtained using the two methods, we can perform dynamic tuning first to select a pool of configurations, and then perform static tuning only on the selected pool.

- **Gray-code Ordering** Our study of the properties of the configuration space of loop fusion indicates that Gray-code ordering is a good coordinate system for heuristic search methods such as a hill climber with random jumps.

- **Register Pressure Estimate** Our study also demonstrates that the performance of loop fusion is constrained by the resource limit on the underlying architecture. In particular, we showed that register pressure estimate could be an effective heuristic for pruning the configuration space of loop fusion.
Chapter 6

Conclusions

This dissertation presents solutions for two technical problems related to compiling
array syntax for high performance on evolving computer architectures: (1) compiling
single array statements onto various architectures, in particular, scalar processors,
short vector processors, and heterogeneous multi-core processors; (2) applying loop
fusion to the result of scalarizing multiple array statements to further improve the
memory hierarchy performance.

The first problem stems from the semantics of the array statements—“fetch-
before-store”. Semantically executing them would require a vector machine whose
vector length is equal to that of array sections. To preserve the semantics when comp-
piling to a variety of real machines which do not support arbitrary vector lengths,
temporary arrays often need to be allocated. Since the allocated temporary arrays
induce extra memory accesses and may cause cache problems in the memory hier-
archy because of a larger memory footprint, we need strategies to avoid their uses
or minimize their sizes. We also need to make sure that the temporary reduction
strategies do not prevent us from exploiting the power of parallelism on the short
vector processors and multi-core processors.

The second problem originates in the limitations of the expressiveness of array syntax. Array syntax cannot naturally express data reuse in the memory hierarchy across array statements. Hence an application of loop fusion on the loop nest obtained by scalarization can bring data reuse among different array statements together. However, the number of possible loop fusion configurations is huge, and static performance models may not derive the best one. We need strategies to search for the best loop fusion configuration in an empirical performance tuning framework.

The solutions we present for these two problems demonstrates that the performance of programs written in array syntax can be significantly improved upon the state-of-the-art implementation on a variety of architectures, including the emerging multi-core processors. In other words, the gap between the expressiveness of array syntax at the high level and the achieved performance on real machines at the low level can be effectively bridged by compiler strategies.

6.1 Contributions

The primary contributions of this dissertation are the following:

1. A scalarization strategy with loop alignment and loop skewing that minimizes temporary array allocation for scalar processors The reduced temporary allocation translates to fewer memory accesses and more efficient use of the memory hierarchy.

2. An extended scalarization strategy for short vector processors The loop alignment algorithm is extended to exploit the vector parallelism, while at
the same time reducing the temporary allocation. In an integrated approach, it is also used to increase the vector data alignment, an important factor for performance on the architectures we target. Besides array statements, the developed vectorization tool also vectorizes general loop nests that can be considered as the results of scalarization from array statements.

3. **An extended scalarization strategy for CELL processor**  We show that loop alignment algorithm for reduced temporary allocation does not prohibit parallelization across processing elements. Moreover, a variation of the loop alignment algorithm, *reduced lazy loop alignment*, can increase the opportunities for parallelization in a loop nest after temporary allocation. On CELL processor, the reduced temporary allocation also translates to fewer data transfers between the SPEs’ LS memory and the main memory.

4. **A dependence-based code generation strategy for a CELL processor**

The tool we developed performs automatic parallelization, vectorization, data alignment, multi-buffering data movement, and synchronization generation.

5. **A strategy to parameterize loop fusion and a performance tuning framework**  Our study suggests that dynamic tuning of loop fusion followed by static tuning only on the configurations chosen by the dynamic fusion can speed up the tuning process. Our analysis of the properties of the configuration space of loop fusion indicates that Gray-code ordering could be a good coordinate system for heuristic search and that constrained resources such as register pressure estimate could be a good candidate for search space pruning.
6.2 Future Work

The strategies and infrastructures developed in this dissertation enable us to explore many interesting ideas as future work for compiling high level program constructs such as array statements and loop nests on the evolving microprocessors with increasing on-chip parallelism.

Pipelined Workflow Our current parallelization strategy on CELL is to identify one parallel loop and partition its iterations across multiple processing elements. One could also construct a task flow graph and then map each node to one or more processing elements. Data flow among the nodes would be converted to data flow among processing elements.

Load Balancing for Heterogeneity Our experiments in Chapter 4 show that a program runs slower on PPE than on SPE on a CELL processor. One could construct a relative performance model for PPE and SPE and use it to assign workload on different PEs. Since the compiler we developed provides an option that specifies the relative ratio between the PPE workload and the SPE workload, we could also determine the best ratio empirically.

Data Reuse in Local Memory Data reuse in the explicitly controlled LS memory of a CELL processor’s SPEs can significantly improve an application’s performance. When compiling array statements, we harnessed the data reuse only within the reference groups. As future research, we would like to investigate the effectiveness of traditional compiler optimizations for improving data reuse such as loop tiling, loop fusion and unroll-and-jam.

Loop Fusion and Synchronization The classic typed-fusion algorithm will partition loops into parallel loops and sequential loops. And for the sake of the
parallelism, sequential loops are fused with parallel loops. If we do fuse a parallel loop with a sequential loop, the resulting loop will be made sequential as well. In our code generation scheme for CELL, we use synchronization to ensure correctness of the sequential loops. Moreover, if the innermost loop only carries anti-dependence with no cycles/recurrence, we can use post-store plus proper synchronization to parallelize it. The question is: Do we fuse for data reuse or for pure parallelism as conventional wisdom? There is obviously an interesting tradeoff for us to determine.
Bibliography


[15] Chun Chen, Jacqueline Chame, and Mary Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *In-
ternational Symposium on Code Generation and Optimization (CGO’05), March 2005.

[16] P. Colella and P.R. Woodward. The piecewise parabolic method for gas-


[17] Computer Science Department, University of Washington. ZPL.


vast.altivec.html.


Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra

Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing

compiler for a CELL processor. In PACT, 2005.


[22] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel R.

Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J.

Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In


