Michigan Technological University
Digital Commons @ Michigan Tech

Dissertations, Master's Theses and Master's Reports - Open
Dissertations, Master's Theses and Master's Reports

2002

Loop transformations for clustered VLIW architectures

Yi Qian
Michigan Technological University

Follow this and additional works at: https://digitalcommons.mtu.edu/etds

Copyright 2002 Yi Qian

Recommended Citation
Qian, Yi, "Loop transformations for clustered VLIW architectures", Dissertation, Michigan Technological University, 2002.
https://digitalcommons.mtu.edu/etds/181

Follow this and additional works at: https://digitalcommons.mtu.edu/etds

LOOP TRANSFORMATIONS FOR CLUSTERED VLIW ARCHITECTURES

By

YI QIAN

A DISSERTATION

Submitted in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

(Computational Science and Engineering)

MICHIGAN TECHNOLOGICAL UNIVERSITY

2002

The dissertation, "LOOP TRANSFORMATIONS FOR CLUSTERED VLIW ARCHITECTURES", is hereby approved in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY in the field of Computational Science and Engineering.

DEPARTMENT or PROGRAM Computational Science and Engineering

Dissertation Advisor:	Dr. Steven M. Carr
Committee:	Dr. Philip Sweany
	Dr. Jean Mayo
	Dr. Stephen A. Hackney
Program Director:	Dr. Phillip Merkey
Date:	August 2002

LOOP TRANSFORMATIONS FOR CLUSTERED VLIW ARCHITECTURES

Copyright 2002
by
YI QIAN

Abstract

LOOP TRANSFORMATIONS FOR CLUSTERED VLIW ARCHITECTURES

by

YI QIAN

Advisor: Dr. Steven M. Carr

With increasing demands for performance by embedded systems, especially by digital signal processing (DSP) applications, embedded processors must increase available instruction-level parallelism (ILP) within significant constraints on power consumption and chip cost. Unfortunately, supporting a large amount of ILP on a processor while maintaining a single register file increases chip cost and potentially decreases overall performance due to increased cycle time. To address this problem, some modern embedded processors partition the register file into multiple low-ported register files, each directly connected with one or more functional units. These functional unit/register file groups are called *clusters*.

Clustered VLIW (very long instruction word) architectures need extra copy operations or delays to transfer values among clusters. To take advantage of clustered architectures, the compiler must expose parallelism for maximal functional-unit utilization, and schedule instructions to reduce intercluster communication overhead.

High-level loop transformations offer an excellent opportunity to enhance the abilities of low-level optimizers to generate code for clustered architectures. This dissertation investigates the effects of three loop transformations, i.e., loop fusion, loop unrolling, and unroll-and-jam, on clustered VLIW architectures. The objective is to achieve high performance with low communication overhead. This dissertation discusses the following techniques:

Loop Fusion This research examines the impact of loop fusion on clustered architectures. A metric based upon communication costs for guiding loop fusion is developed and tested on DSP benchmarks.

Unroll-and-jam and Loop Unrolling A new method that integrates a communication cost model with an integer-optimization problem is developed to determine unroll

amounts for loop unrolling and unroll-and-jam automatically for a specific loop on a specific architecture.

These techniques have been implemented and tested using DSP benchmarks on simulated, clustered VLIW architectures and a real clustered, embedded processor, the TI TMS320C64X. The results show that the new techniques achieve an average speedup of 1.72-1.89 on five different clustered architectures.

Acknowledgements

I would like to thank my advisor, Dr. Steve Carr, for his inspiration and continuous support of my PhD research. Dr. Steve Carr is an excellent advisor who is always willing to listen, encourage, and give insightful comments and valuable criticism. He read all the drafts of my papers and dissertation and taught me to be thorough in analyzing problems and rigorous in presenting ideas. This dissertation would not have been possible without his support and guidance. I am also greatly indebted to my previous advisor, Dr. Phil Sweany, for bringing me into the field of compiler optimizations about four years ago. Since then I have benefited from discussions with Dr. Sweany and the valuable comments that he has offered. Special thanks also goes to my committee members, Dr. Jean Mayo and Dr. Stephen Hackney, for their helpful suggestions and hard questions during the course of developing this dissertation.

My summer internships gave me a chance to work with intelligent engineers and researchers in industry, to connect my academic knowledge with practice, and to shape the ideas in this research work. I am grateful to all the colleagues who were so helpful to me when I interned at Starcore and Texas Instruments during the summer of 2000 and 2001. Additionally, I want to thank StarCore for providing support in the initial stages of this work. I also appreciate Texas Instruments for providing the benchmark suite used in this work and Dr. John Linn for his help in obtaining the benchmarks.

I want to express my gratitude to my former graduate advisor, Prof. Tongsheng Wang, at Tianjin University (P. R. China) for being supportive and helpful when I applied for PhD programs in the US. In the past few years, the conversations we had through phone calls, letters, and emails have been a source of encouragement for me as I faced difficulties during the course of accomplishing my degree.

I must thank Dr. Denise Heikinen for help with editing the draft of this dissertation. Also I have been supported by the National Science Foundation and a finishing fellowship from the Graduate School of Michigan Technological University.

Finally, I wish to thank my parents for giving me life and intelligence, and for teaching me to be honest, diligent, and persistent. Their everlasting love and support have been the source of my strength and the reason why I have come this far.

Contents

List of Tables iii

List of Figures iv

1 Introduction 1

1.1 Background . . . 3

1.1.1 Data Dependences . . . 3

1.1.2 Machine Balance and Loop Balance . . . 6

1.1.3 ILP and Clustered VLIW Architectures . . . 6

1.1.4 Compiler Techniques for Clustered VLIWs . . . 11

1.2 Related Work . . . 19

1.2.1 Partitioning Problem . . . 20

1.2.2 Loop Transformations . . . 21

2 Loop Transformation Strategy 26

2.1 Compiler Structure . . . 26

2.2 Examples . . . 28

2.3 Loop Transformation Strategy . . . 33

2.4 Overhead . . . 33

3 Unroll-and-jam 35

3.1 Safety and Updating the Dependence Graph . . . 35

3.2 Determining Loops to Unroll . . . 38

3.3 Computing *unit MinII* . . . 39

3.3.1 Unrolling a Single Loop . . . 40

3.3.2 Unrolling Multiple Loops . . . 46

3.4 Register Pressure . . . 50

3.5 Computing Unroll Amounts . . . 52

3.6 Summary . . . 55

4 Loop Fusion 56

4.1 Safety of Loop Fusion . . . 56

4.2 Communication Cost . . . 58

4.3 Implementing Fusion . . . 60

4.4 Summary 61

5 Experimental Results **62**

5.1 Benchmarks and Configurations 62

5.2 Unroll-and-jam Results 64

5.2.1 URM Results 64

5.2.2 TMS320C64x Results 67

5.2.3 Accuracy of Communication Cost Prediction 68

5.3 Fusion results 68

5.3.1 URM Results 69

5.3.2 TMS320C64x Results 71

5.4 Summary 72

6 Conclusions and Future Work **73**

6.1 Contribution 73

6.2 Future Work 74

6.3 Final Remarks 76

A Individual Loop Performance for Unrolling **77**

B Individual Loop Performance for Loop Fusion **94**

Bibliography **101**

List of Tables

5.1 Operation Cycle Counts . . . 63
5.2 URM Speedups: Unrolled vs. Original . . . 65
5.3 URM Speedups: Transformed vs. Original . . . 65
5.4 URM Speedups: The New Algorithm vs. Fixed Unroll Amounts . . . 67
5.5 TMS320C64X Speedups: Unrolled vs. Original . . . 67
5.6 Loop Description . . . 69
5.7 URM Speedups: Fused vs. Original . . . 69
5.8 URM Speedups: Fused & Unrolled vs. Unrolled Only . . . 70
5.9 URM Speedups: Fused & Unrolled vs. Original . . . 70
5.10 TMS320C64x Speedups: Fused vs. Original . . . 71

A.1 Unrolling Results for Individual Loops on 8-wide 2-cluster . . . 79
A.2 Unrolling Results for Individual Loops on 8-wide 4-cluster . . . 81
A.3 Unrolling Results for Individual Loops on 16-wide 2-cluster . . . 83
A.4 Unrolling Results for Individual Loops on 16-wide 4-cluster . . . 85
A.5 Unrolling Results for Individual Loops on TMS320C64x . . . 88
A.6 The New Algorithm vs. Fixed Unroll Amounts . . . 90
A.7 Predicted Copies vs. Intercluter Dependences . . . 93

B.1 Fusion Results for Individual Loops on 8-wide 2-cluster . . . 96
B.2 Fusion Results for Individual Loops on 8-wide 4-cluster . . . 97
B.3 Fusion Results for Individual Loops on 16-wide 2-cluster . . . 98
B.4 Fusion Results for Individual Loops on 16-wide 4-cluster . . . 99
B.5 Fusion Results for Individual Loops on TMS320C64x . . . 100

List of Figures

1.1 User Level Code 7
1.2 Sequential Low-level Code 7
1.3 VLIW Code 8
1.4 Clustered VLIW DSP: The TMS320C64x 10
1.5 Software Pipelining Example 13
1.6 Ideal Schedule 15
1.7 Register Component Graph 16
1.8 Partitioned Schedule 16

2.1 Compiler Structure 27

3.1 Updating the Dependence Graph 39
3.2 Compute Unit Communication Cost for E_l^C 47
3.3 Compute Unroll Amounts 54

4.1 Determine Profitability of Loop Fusion 60

Chapter 1

Introduction

Modern architectures achieve great performance through parallel technology. Parallelism ranges from fine-grained parallelism such as instruction-level parallelism(or ILP) to coarse-grained parallelism such as process-level parallelism. ILP architectures, e.g., superscalar and VLIW (very long instruction word), gain computation speedups via simultaneously executing multiple low-level instructions in different functional units. This relies on complex hardware or advanced compilers to extract independent instructions from sequential code. In superscalar architectures, several dynamically scheduled instructions can be issued in a single cycle, while in VLIW architectures, independent instructions, called operations, are detected and grouped into a very long word instruction at compilation time such that a multi-operation instruction can be issued in each cycle. Compiler techniques, such as software pipelining and global instruction scheduling, have been proven necessary methods to increase the degree of instruction-level parallelism in programs.

A high degree of ILP requires a large register file with a good number of read/write ports to support simultaneous access to registers, because typically one write port and two read ports are needed for each functional unit. Considering that the current architecture techniques usually restrict the number of read/write ports to fewer than 20 in a single register file, the number of functional units connected to a register file is limited [57]. Furthermore, an increase in the number of read/write ports may impede overall performance due to the increased access time. As a result, supporting a high degree of ILP via a single large register file becomes unrealizable and unprofitable.

The search for a high degree of ILP with low demands on register ports gives rise to the introduction of clustered architectures. Clustered VLIW architectures partition

a single register file into multiple small register files with a low number of ports, each of which is associated with one or more functional units. Such functional unit/register file groups are called *clusters*. Each functional unit has direct access to its local register file and limited connection to remote register files. The overhead of clustered architectures comes from the additional data transfers across clusters when a functional unit needs values from other clusters. To take advantage of clustered VLIW architectures, the compiler must both expose more parallelism for maximal functional-unit utilization, and schedule instructions among clusters such that the overhead caused by intercluster communication is minimized.

Some of state-of-the-art embedded processors, in particular DSP chips (e.g., the Texas Instruments TMS320C6x), have employed clustered VLIW architectures. Embedded system applications typically are sensitive to memory usage, and energy consumption, as well as performance. Therefore, sophisticated compiler techniques are needed to achieve high performance while maintaining modest power consumption and code size on clustered VLIW architectures.

Much recent work in compilation for clustered VLIW architectures has concentrated on methods to partition virtual registers amongst the target architecture's clusters effectively for software pipelined loops [4, 32, 48, 66]. Since it is commonly accepted that general applications spend a large fraction of time in loops, high-level loop transformations offer an excellent opportunity to enhance these partitioning schemes and greatly improve performance.

This dissertation shows how to utilize loop transformations to reduce communication overhead and exploit good ILP for architectures with partitioned register files. Particularly, the effect of three loop transformations, i.e., loop fusion, unroll-and-jam, and loop unrolling, is investigated. New metrics based upon communication cost models are developed to guide loop transformations. The experimentation shows the optimization algorithms proposed in this dissertation are effective in enhancing software pipelining and generating efficient code.

This dissertation consists of six chapters. Chapter 1 introduces the background related to ILP architectures and compiler techniques for clustered VLIW machines and summarizes the previous work. Chapter 2 gives an overview of the loop transformation strategy used in this work. Chapter 3 discusses loop unrolling and unroll-and-jam to improve ILP with low overhead. Chapter 4 addresses loop fusion to enhance parallelism on clustered architectures. Chapter 5 reports the experimental results. Finally, Chapter 6 concludes and

offers suggestions for future work.

1.1 Background

This section is devoted to presenting a foundation for understanding this dissertation. First, we describe the concept of dependence to aid in the discussion of the legality and profitability of reordering transformations. Then, a measure of machine and loop performance is introduced. Next, we outline the advantages and limitations of clustered VLIW machines. Finally, compiler techniques for clustered architectures are presented.

1.1.1 Data Dependences

A transformation is *legal* (or *safe*) if the transformed code preserves the program semantics. This research is based upon dependence analysis to determine the safety and profitability of a transformation.

There is a dependence between two references R_1 and R_2 if both references access the same memory location, and there is a feasible run-time execution path from R_1 to R_2 [40]. Dependence represents a relation between the references in a program. If a program is executed in sequential order, dependences do not cause problems. Dependences impose constraints on the execution order of instructions if the program is transformed to achieve parallel execution. More specifically, only if the dependence information is detected and preserved can the transformed program compute the same result as the original program. For instance, a loop transformation that reorders the execution of statements in a loop is legal if and only if the resulting loop satisfies the dependence constraints.

Data dependences can fall into three categories [40, 41]:

1. *true dependence* - the first reference stores into a location that is later used by the second reference.

2. *antidependence* - the first reference reads from a location into which the second reference later stores.

3. *output dependence* - the two references write into the same location.

Program transformations must observe these dependences in order to preserve the semantics of the original program. There exists another dependence, termed *input*

dependence, that occurs when two references read from the same location. Reordering two instructions with input dependences does not affect the meaning of the program. Input dependences are, however, useful for detecting and improving data reuse.

Data dependences are usually represented by a data dependence graph (DDG) where each node is a reference (or the statement that contains this reference) and the edges represent the dependences between two nodes.

Dependences in loops can be further classified into two categories: *loop independent* and *loop carried* [10]. A dependence is *loop independent* when control flows from the first statement to the second within a single iteration of the loop. For example, the dependence between two references to `A(I)` in the following loop is loop independent:

```
Do I = 1, N
   A(I) = B(I) + X
   C(I) = A(I)
ENDDO
```

Such a dependence exists on every iteration of the `I`-loop.

A dependence is *loop-carried* if control flows from the first statement to the second crossing an iteration of the loop, as the dependence between `A(I)` and `A(I-1)` in the following loop shows

```
DO I = 1, N
  A(I) = B(I) + X
  C(I) = A(I-1)
ENDDO
```

`A(I)` and `A(I-1)` will not refer to the same location unless the loop is iterated. Consider another loop as follows:

```
DO I = 1, N
  DO J = 1, N
    A(I,J) = B(I,J) + X
    C(I,J) = A(I-1,J)
  ENDDO
ENDDO
```

the dependence from `A(I,J)` to `A(I-1,J)` exists when the outer loop is iterated. In this case we say that the dependence is *carried* by the outer loop (`I` loop).

A loop-independent dependence is preserved if the original statement order is preserved. A loop-carried dependence, however, is strongly dependent upon the order in which the loops are iterated, but independent of statement order within the loops.

Dependences in loops can be represented via *distance vectors* [42] or *direction vectors* [65]. To define distance and direction vectors, a representation of a specific loop iteration, termed *iteration vector* is first given. An iteration vector $\vec{i}$ is simply a vector of values for the loop control variables. The set of iteration vectors corresponding to all iterations of the loop nest is called the *iteration space*. Using iteration vectors, we can define the *distance vector* and *direction vector*. If two iteration vectors $\vec{i}$ and $\vec{j}$ represent the execution of two references that are contained in n common loops, then

the *distance vector* (or *distance*) $d(i, j)$ between the references is defined as a vector of length n, such that the kth component of the distance vector $d(i, j)_k$ is equal to $\vec{j}_k - \vec{i}_k$.

the *direction vector* $D = (i, j)$ is defined as a vector of length n, such that the kth component of the direction vector is

$$D(i,j)_k = \begin{cases} <, & \text{if } d(i,j)_k > 0 \\ >, & \text{if } d(i,j)_k < 0 \\ =, & \text{if } d(i,j)_k = 0. \end{cases}$$

For instance, the distance vector between `A(I,J)` and `A(I-1,J)` in the previous example is $\langle 1, 0 \rangle$, and the direction vector is $\langle <, = \rangle$. Goff *et al.* proposed a practical dependence testing scheme for determining distance and direction vectors in [30].

The loop associated with the outermost nonzero direction (or distance) vector entry is called the *carrier* of the dependence. If the value of the distance vector entry for the dependence carrier is constant throughout the execution of the loop, the dependence is called a *consistent dependence*; otherwise it is called an *inconsistent dependence*.

Data dependence analysis provides the information on the execution order constraints of operations in loops. To measure the performance of loops, the notion of *balance*, described below, is used.

1.1.2 Machine Balance and Loop Balance

Callahan *et al.* first introduced the notion of balance to estimate how efficiently a given loop can be executed on a particular ILP machine [14].

A computer is balanced when it can operate in a steady state manner with both memory accesses and floating-point operations being performed at peak speed. *Machine balance*, β_M, is defined as the rate at which operations can be fetched from memory, M_M, compared to the rate at which floating-point operations (*flops*) can be performed, F_M:

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

Similarly, balance for a loop L is defined as

$$\beta_L = \frac{\text{number of words accessed} = M_L}{\text{number of flops performed} = F_L}.$$

These two metrics give us a measure of the performance of a loop executed on a particular machine. If $\beta_L > \beta_M$, the loop needs data at a higher rate than the memory system can provide and idle computational cycles will exist. Such a loop is called a *memory-bound* loop. If $\beta_L < \beta_M$, more memory operands can be delivered in a time unit than the floating-point unit can process. In this case the loop is called a *compute-bound* loop. If $\beta_L = \beta_M$, the loop is balanced and runs well on that machine.

1.1.3 ILP and Clustered VLIW Architectures

Parallel processing has emerged as a solution to the increasing demand for high speed and low cost in today's computer applications. Parallelism appears in various forms, among them is instruction-level parallelism, a fine-grained parallelism. Unlike the coarse-grained parallel processing which allows large sections of code to be run in parallel on independent processors, ILP utilizes the parallel execution of the lowest level computer operations to increase performance. ILP architectures gain computation speedup at a cost of increased demand on register resources. Some state-of-art embedded system processors use clustering to simplify hardware design and support a high degree of ILP.

This section introduces ILP architectures and discusses advantages and limitations of clustered VLIWs.

1.1.3.1 Introduction to ILP

In ILP architectures, multiple instruction-level operations, like additions, multiplications, loads, etc., are issued to multiple functional units during each machine cycle [53]. Consider, for example, the user-level code in Figure 1.1 for which the sequential low-level operations are shown in Figure 1.2:

```
e = a * b + b / 2
f = c - d + b
g = e + f
```

Figure 1.1: User Level Code

Suppose each addition or subtraction operation takes one cycle to complete, and each multiplication or division operation takes four cycles. If these operations were executed in sequential order, then a total of 12 cycles would be needed. However, the sequential execution is not necessary if the architecture provides multiple functional units. Operations that do not depend on one another can be executed simultaneously in different functional units. The first, the second, and the fourth operations in this example have no dependences; therefore, their execution can be overlapped. This is also true for the third and the fifth operations. The third operation must follow the first two operations, since it needs the values calculated by the first two operations. Thus by executing some operations in parallel, the total cycles can be reduced by half (see Figure 1.3).

The use of ILP is transparent to the programmers, who do not have to change their algorithms or programs. This has presented challenges to processors and compilers which are expected to be in charge of exploiting ILP available for the target architecture.

```
r1 = a * b
r2 = b / 2
e = r1 + r2
r3 = c -d
f = r3 + b
g = e + f
```

Figure 1.2: Sequential Low-level Code

r1 = a * b	r2 = b / 2	r3 = c - d
nop	nop	nop
nop	nop	nop
nop	nop	nop
e = r1 + r2	f = r3 + b	
g = e + f		

Figure 1.3: VLIW Code

1.1.3.2 Superscalar and VLIW

Two types of ILP architectures are superscalar and VLIW, representing two methods of scheduling parallel instructions through hardware and software solutions. Both superscalar and VLIW provide multiple functional units and aim to speed up computation by making efficient use of concurrently working functional units. The differences reside in how the instructions are constructed and how the dependences are detected. In superscalar processors, specialized hardware is used to detect dependences dynamically and determine the issuing order. Parallelism in superscalar processors comes from issuing multiple short, RISC-like instructions in a cycle. On the other hand, VLIW machines rely on compilers to extract parallelism and generate a stream of long word instructions, each consisting of multiple fields that specify the operations for each functional unit. When no operation is found for a functional unit, a nop operation is explicitly inserted. Figure 1.3 shows the VLIW code of Figure 1.1 on a VLIW processor with 3 functional units.

The dynamic features of a superscalar processor leads to unpredictable execution time and increased power consumption, making it a poor candidate for DSP applications. The execution time in superscalar processors may vary during different executions of a program based on the accessed data [26]. For example, the processor may select independent instructions to be executed in parallel one way the first time a program is executed, but issue the parallel instructions in another way the next time the same program is executed. This raises difficulties for programmers, since programmers of real-time applications must predict how long it will take a piece of code to complete. Measuring execution time using the worst-case implementation is a solution; however, it may waste machine resources and hence cause performance degradation. In addition, complicated hardware in superscalar processors further increases constraints of power consumption of DSP applications. As a

result most commercial DSP chips (e.g., TI's TMS320C6000 and Starcore's SC140) have adopted VLIW architectures.

1.1.3.3 Clustered VLIW Architectures

To perform efficiently, superscalar and VLIW architectures depend on activating as many simultaneous functional units as possible. The compiler techniques described in Section 1.1.4 have been used to extract a high level of ILP from programs run on a VLIW machine. However a high degree of parallelism gives rise to a high demand on machine resources, such as registers, read/write ports, and buses, which in turn may impede performance. Consider an *ideal* VLIW machine where multiple functional units share the use of a common register file, typically a functional unit must, in a single cycle, be able to read two operands from the register and write the result to the register. To support a high degree of parallelism, multiple functional units have to be satisfied with concurrent accesses to a good number of read/write ports in each cycle. From a technological perspective, it is impractical or inconvenient to build a single register file with an excessive number of ports. Furthermore, Capitanio *et al.*, showed via their experiments that the access time is roughly proportional to the logarithm of the number of output ports [15]. As a result, the overall performance may be hampered as the number of functional units increases. This means that the number of functional units in an ILP machine is limited due to the constraints on the chip space and access time.

Many state-of-art DSP architectures have utilized clustering as a solution to achieve a high degree of ILP as well as a high clock rate. In clustered VLIW machines, the register file is partitioned into several separate register files with a small number of read/write ports. Each register file is grouped with one or more functional units such that a functional unit has direct access to the local register file. These register file/functional units groups are called *clusters*. For example, TI's TMS320C64x has two register files, each associated with four functional units, as shown in Figure 1.4 [60]. Thus each register file only needs to support 12 ports.

A mechanism must be provided to enable a functional unit to access values residing in other register files. One possible approach is to use an interconnection network that connects each functional unit to each register file. This method provides a fast means to access data in the remote clusters, but is not feasible for a large number of functional units

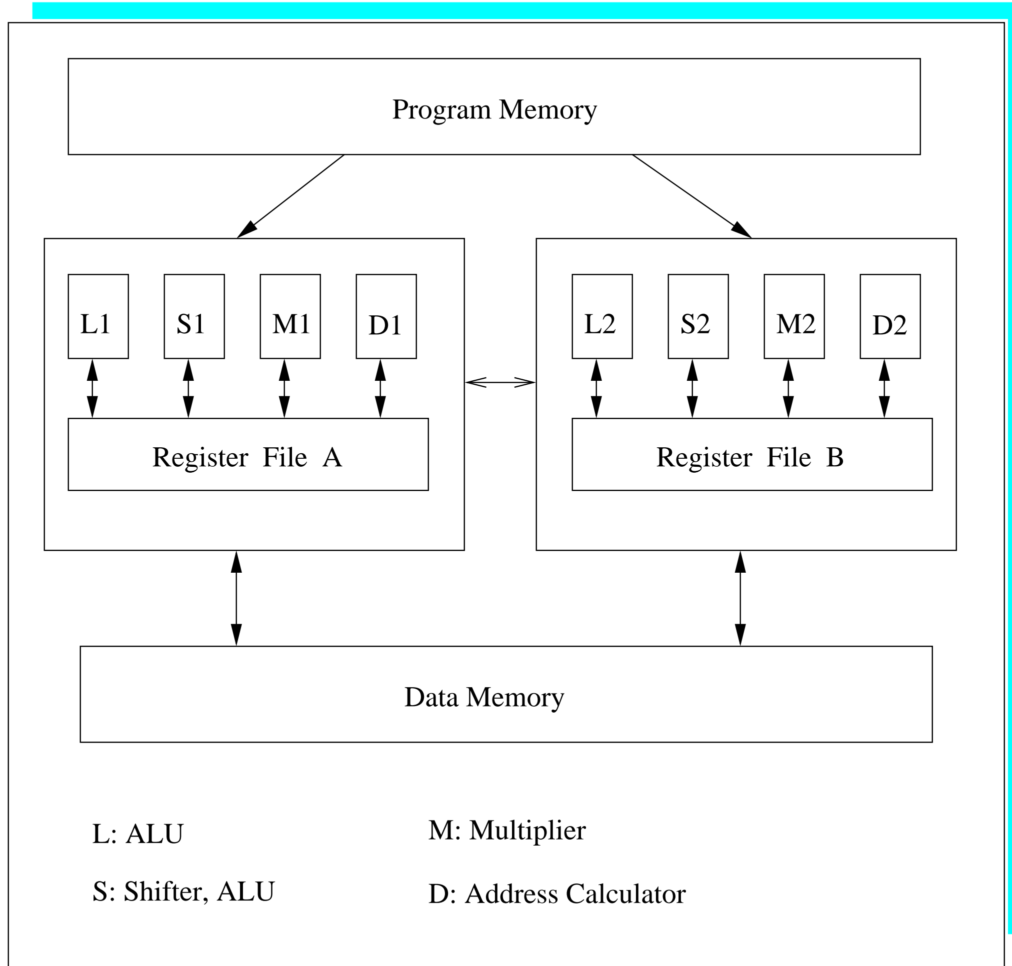

Figure 1.4: Clustered VLIW DSP: The TMS320C64x

when complexity and cost are taken into account. An alternative approach is to transfer data among clusters via explicit value copy operations, moving values through a data bus shared by all functional units. In this case, it is the compiler's job to schedule instructions among clusters and insert value copy operations when needed. These copy operations require extra execution time and hardware resources, and, thus, may degrade performance over the ideal but not realizable VLIW architectures. The goal of a compiler is to minimize the overhead caused by additional copy operations, while maintaining a high degree of ILP. To gain the best performance, the compiler must trade off between two scheduling schemes: the distribution of operations among all available functional units to obtain the maximum parallelism and the use of as fewer clusters as possible in order to minimize intercluster communication overhead.

1.1.4 Compiler Techniques for Clustered VLIWs

Since instruction parallelism in a VLIW machine is completely determined at compilation time, the compiler plays a very important role in performance. Two of the main tasks of a clustered VLIW complier are *instruction scheduling* and *register partitioning*. In the instruction scheduling phase, the compiler reorders the execution of machine operations to exploit parallelism (Section 1.1.4.1). Particularly a technique termed *software pipelining* has been used to enhance parallelism of loops (Section 1.1.4.2). In the register partitioning phase, the values are spread among clusters to achieve both maximum parallelism and minimum intercluster communication (Section 1.1.4.3).

High-level transformations are effective ways to enhance the ability of a compiler to produce high-quality code for clustered machines (Section 1.1.4.4). Some loop transformations, e.g., scalar replacement, loop fusion, and unroll-and-jam, can be applied to improve utilization of hardware resources in a single cluster. Others such as loop unrolling and unroll-and-jam allow a high degree of intercluster parallelism to be achieved.

It is hard for a compiler to generate excellent code due to interaction of these optimizations. Therefore, it is essential to study the effects of each individual technique as well as the tradeoff that must be made when using multiple optimizing techniques in order to maximize the overall benefit.

1.1.4.1 Instruction Scheduling

The compiler rearranges the ordering of machine operations in order to effectively use an ILP architecture's parallelism. This technique is known as *instruction scheduling*. Building an optimal scheduling of instructions has been proven to be a NP-complete problem [23] and several scheduling methods have been developed to generate suboptimal code.

List scheduling is the most popular scheduling technique to increase parallelism within a single basic block[1] [28]. List scheduling iteratively selects an operation from a list of ready-to-execute operations and schedules it if there exists no resource conflicts. An operation is ready-to-execute if it is dependence free (when all source operands have valid values). One or more heuristics can be used to determine the priorities for each operation and to break ties when several operations have the same priority.

However, basic block scheduling, or local scheduling, cannot extract sufficient parallelism to feed ILP processors with a large number of functional units. It has been shown that the average number of parallel operations that are restricted by true dependences in a basic block ranges from one to three, indicating the potential speedup of an ILP processor over a conventional processor is two [42, 44, 36, 62]. There are two means by which we can achieve a higher degree of ILP: software pipelining and global scheduling. Software pipelining exploits parallelism in loops, while global scheduling extracts as many independent operations as possible from code in multiple basic blocks. *Trace scheduling* [27] and *dominator-path scheduling* [58] are two global scheduling techniques. Trace scheduling iteratively selects the most frequently executed path from the unscheduled code of a program (such a path is called a *trace*), and schedules it as though it were a single block. Dominator-path scheduling performs scheduling on a group of basic blocks based upon the dominator [2] relation. Unlike most global scheduling methods, dominator-path scheduling does not require copies of operations to preserve the program semantics. Other global scheduling techniques are described in [3], [11], and [31]. Local scheduling is simple to apply, while global scheduling is more effective at using available parallelism in that it schedules instructions beyond the block boundaries.

[1] A basic block is a single entrance, a single exit sequence of operations that can have a branch only at the bottom.

[2] A block B_1 dominates block B_2 if and only if every path from the program entry to B_2 contains B_1.

1.1.4.2 Software Pipelining

Software Pipelining is a loop transformation technique to achieve instruction-level parallelism [5, 45, 52]. Software pipelining gets its name from its similarity with hardware pipelining, where the execution of different instructions is overlapped. In software pipelining the execution of different loop iterations can be overlapped.

A simple example to illustrate software pipelining is shown in Figure 1.5. Suppose the code is run on a VLIW machine with two functional units, and each operation takes one cycle to complete. In this figure, we use the convention that X^m is the version of statement X on the mth iteration. The simple local scheduling can not schedule two additions in the same cycle due to the true dependence between them. Thus, up to 200 ($2\times100 = 200$) cycles are necessary to complete the loop. However parallelism can be exploited when we find that A^2 can actually follow the execution of A^1 immediately and be run at the same cycle with B^1 (as shown in the second row in Figure 1.5(b)). This pattern can be repeated such that two operations (from two successive iterations) are started each cycle. The instructions of a repeating pattern are called the *kernel* of the software pipelined loop. The pipelined code shown in Figure 1.5 (c) needs 101 ($1 + 1 \times 99 + 1 = 101$) cycles to complete, achieving the approximate speedup of 2.

```
DO I = 1, 100
    A: A(I+1) = A(I) + 2
    B: B(I) = A(I+1) + B(I)
ENDDO
```

(a) Original Loop Code

A^1
A^2 B^1
A^3 B^2
....
A^{100} B^{99}
B^{100}

(b) Schedule

```
A(2) = A(1)+2
DO I = 1, 99
   A(I+2) = A(I+1) + 2 || B(I) = A(I+1) + B(I)
ENDDO
B(100) = A(101) + B(100)
```

(c) Software Pipelined Code

Figure 1.5: Software Pipelining Example

The delay between the initiation of iterations of the pipelined loop is termed the *initiation interval*, or II [5]. For instance, the II of the pipelined loop in Figure 1.5 is one. The goal of software pipelining is to minimize II.

This work uses a software pipelining approach called *modulo scheduling* [52]. Modulo scheduling selects a schedule for one iteration of a loop such that, when that schedule is repeated, no resource or dependence constraints are violated. To do this, modulo scheduling starts by predicting the minimal number of instructions required between initiating execution of successive loop iterations, called the *minimum initiation interval* or *MinII.* When determining the MinII, modulo scheduling considers two factors, the constraints from the machine resources, termed the *resource initiation interval* or *ResII* , and the limitations from the dependences between operations, termed the *recurrence initiation interval* or *RecII.* ResII is the maximum number of instructions in a loop requiring a specific functional unit resource, and RecII is the length of the longest recurrence in the data dependence graph of a loop. The maximum of RecII and ResII imposes a lower bound on MinII. Once MinII is determined, instruction scheduling attempts to find a schedule in MinII instructions. If no such schedule can be found, then the II is incremented and software pipelining is tried again. Otherwise, code to set up the software pipeline (prelude) and drain the pipeline (postlude) are added.

1.1.4.3 Register Partitioning

A compiler in a clustered VLIW machine has to achieve both a maximum level of parallelism by using aggressive instruction scheduling and a minimum amount of intercluster communication by wisely distributing the data among partitioned register files. The technique used in this work to partition registers for a clustered VLIW machine is called *register component graph partitioning* [35]. This method builds a *register component graph* (RCG) from an "ideal" instruction schedule, a schedule for an equivalent VLIW machine with a single multi-ported register file. In the RCG, each node represents a register operand (symbolic registers) and edges are added between the destination registers and the source registers for each (atomic) operation. The graph is partitioned using a greedy heuristic based upon the weights associated with the nodes and edges. The nodes that are connected by an edge with a positive weight should be placed into the same register file; otherwise, they should go in separate clusters. The edge and node weights are based upon schedul-

ing freedom, parallelism and copy costs. After partitioning the RCG, the instructions are re-scheduled based upon the register locations and copies are inserted when necessary.

To demonstrate the register component graph method of partitioning, consider the following example:

```
DO I = 1, 2*N, 2
  S1 = 0
  S2 = 0
  DO J = 1, N
    B = X(J)
    K = I + J
    S1 = S1 + A(K) * B
    S2 = S2 + A(K+1) * B
  ENDDO
  C(I) = S1
  C(I+1) = S2
ENDDO
```

We software pipeline the inner loop, assuming a single cycle latency for addition, and a two-cycle pipeline for multiplication. In addition, it is assumed that all memory accesses are cache hits and also use a two-cycle pipeline. An ideal schedule for this inner loop in a 2-wide VLIW machine is shown in Figure 1.6 (The symbols represent registers). The corresponding register component graph appears in Figure 1.7. The loop kernel requires 4 cycles to complete.[3]

mult t1, a1, b	add k, i, j
mult t2, a2, b	load a1, A(k++)
add s1, t1, s1	load b, X(j++)
add s2, t2, s2	load a2, A(k)

Figure 1.6: Ideal Schedule

One potential partitioning of the graph in Figure 1.7 (given the appropriate edge and node weights) is the following:

$$P_1 = \{t1, a1, s1, b, i1, j1\},\ P_2 = \{t2, a2, s2, k, i2, j2\}$$

[3]We assume the availability of a load command that allows for either pre-increment or post-increment of the address operand.

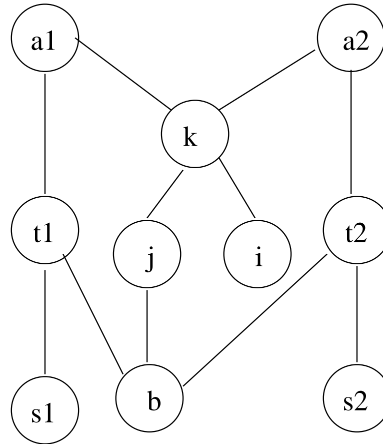

Figure 1.7: Register Component Graph

mult t1, a1, b	add k, i, j
load b, X(j++)	add s2, t2, s2
copy k1, k	load a2, A(++k)
load a1, A(k1)	copy b2, b
add s1, t1, s1	mult t2, a2, b2

Figure 1.8: Partitioned Schedule

In the above partition we assume that `i` and `j` (read-only variables) are cloned to have one copy in each of the clusters [43]. This assumption allows us to produce the partitioned code of Figure 1.8, which requires 5 cycles for the loop kernel, a degradation of 20% over the ideal schedule of Figure 1.6. The additional instructions are necessary to copy the value of `k` from cluster 2 to cluster 1, where it is known as `k1`, and to copy the value of `b` from cluster 1 to cluster 2, where it is known as `b2`.

1.1.4.4 Loop Transformations

Scalar Replacement Scalar replacement [14] transforms array references into sequences of temporary scalar variables to achieve array element reuse. Frequently accessed array references, after replaced with scalars, are likely to be assigned into registers; hence fewer memory accesses are needed.

For example, in the loop

```
DO I = 1, N
  A(I) = 0;
  DO J = 1, N
    A(I) = A(I) + B(J) * C(I, J)
  ENDDO
ENDDO
```

the reference to `A(I)` can be moved out of the inner loop and replaced with a scalar temporary as follows:

```
DO I = 1, N
  T = 0;
  DO J = 1, N
    T = T + B(J) * C(I, J)
  ENDDO
  A(I) = T
ENDDO
```

One memory load and one memory store are removed from the innermost loop via scalar replacement. On most architectures the second loop would run faster.

Loop Fusion Loop fusion, or loop jamming [6], combines loop bodies of multiple countable loops that have the same loop limits into a single loop. Fusion can reduce loop overhead and expose more instructions for local optimization. More importantly, loop fusion improves data reuse by bringing references to the same memory location closer together. Consider the loops

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

Fusing these loops would result in

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

The fused loop can capture temporal reuse for the array `A` and `B`. Chapter 4 will investigate the impact of loop fusion on clustered VLIW machines.

Loop Unrolling and Unroll-and-jam Loop unrolling [6] is a transformation that unrolls the innermost loop to generate several copies of the loop body. Originally loop unrolling was used to reduce inter-iteration overhead (such as testing for loop end and branching back conditionally) and to enlarge loop body size to obtain a better schedule. For clustered VLIW architectures loop unrolling can be employed to improve intercluster parallelism [54, 55].

Unroll-and-jam, or outer-loop unrolling [6], unrolls the outer loop and then fuses the copies of the inner loops. Unroll-and-jam introduces more computations into an innermost loop body without a proportional increase in memory references; therefore, it is used to balance the computation and memory-access requirements of a loop with the corresponding resources provided by a specific machine. For example, applying unroll-and-jam to the loop

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

will produce

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

The original loop has three loads, one store, and two computations, giving a loop balance of 2. After unroll-and-jam, the loop has six memory accesses and four computations, reducing the loop balance to 1.5. This is because the data reuse involving `B(J)` and `C(J)`, which existed at the outer loop level, is moved to the innermost loop after unroll-and-jam. Chapter 3 will discuss how to apply loop unrolling and unroll-and-jam for clustered VLIW machines.

Loop Alignment Loop alignment [7] can convert the loop carried dependences into loop independent dependences, so that the loop computes and uses values on the same iteration. This converts loop-carried dependences into the loop-independent dependences. Alignment is used to improve the parallelism and reduce the register pressure. For example, the following loop has a loop carried dependence involving `A(I)`:

```
DO I = 1, N
  A(I) = B(I) + C(I)
  X(I) = A(I-1) * Q
ENDDO
```

Aligning `A(I)` and `A(I-1)` would result in a loop as follows:

```
X(1) = A(0) * Q
DO I = 1, N-1
  A(I) = B(I) + C(I)
  X(I+1) = A(I) * Q
ENDDO
A(N) = B(N) + C(N)
```

where the definition and use of `A(I)` are on the same iteration. The aligned loop carries no dependences, and hence can be parallelized.

However, loop alignment is not sufficient to eliminate all loop-carried dependences. A situation in which two or more dependences cannot be simultaneously aligned is called an *alignment conflict*. Alignment conflict occurs when recurrences exist or multiple dependences between the same statements have different distances. For instance, the two dependences between the statements in the following loop have distances of 1 and 2 respectively, making it impossible to align them at the same time.

```
DO I = 2, N
  A(I) = B(I) + Q
  C(I) = A(I-1) + A(I-2)
ENDDO
```

More details regarding how alignment conflicts affect loop transformations on clustered architectures can be found in Chapter 2, Chapter 3, and Chapter 4.

1.2 Related Work

Several algorithms have been proposed to deal with instruction assignment and scheduling at compilation time for clustered VLIW architectures. This section gives an overview of previous work solving the partitioning problem and a summary of loop transformations used for various architectures.

1.2.1 Partitioning Problem

The first work regarding partitioning and scheduling code on a multiple register file architecture can be traced back to 1985 when Ellis presented a bottom-up greedy (BUG) algorithm in the bulldog compiler [25]. BUG schedules operations in a collection of basic blocks termed a *trace* according to availability of functional units and registers, and inserts value copy operations if necessary.

Capitanio *et al.* examined performance degradation for a clustered VLIW model called the limited connectivity VLIW (LC-VLIW), and analyzed architectural tradeoffs for LC-VLIW [15]. Their code partitioning method partitions a directed cyclic graph derived from scheduled code for an ideal VLIW machine with a single, multi-ported register file. Value copies are inserted into the partitioned code which is then compacted to generate the final code. Their experimental results suggest good performance can be expected by using partitioned register files with a small number of read/write ports, while limiting the number of intercluster value copies.

Nystrom and Eichenberger proposed an iterative approach to generate effective software pipelined loops for a clustered VLIW machine [48]. Their method consists of two phases: cluster assignment and modulo scheduling. Failure of either phase will lead to restarting the whole process with an increased initiation interval. The cluster assignment phase avoids increasing II in two ways: first it predicts and reserves resources for future copy operations to reduce the likelihood of assignment failure. Secondly it assigns instructions of a recurrence into the same cluster whenever possible to prevent increasing RecII. Their results show that a majority of loops scheduled by this algorithm maintain identical IIs for clustered architectures compared with an equivalent architecture with a single register file.

Desoli exploited a heuristic method addressing instruction assignment for DSP applications with large amounts of computation [22]. This method first pre-partitions instructions to clusters, trying to minimize the schedule length by gathering the nodes in critical paths together. Then the initial partitions are mapped to clusters to get a balanced load. Finally, the resulting code is refined by a simplified list scheduler to further reduce the schedule length. This method proves simple and efficient for the DSP benchmarks which have a large number of instructions and present symmetric patterns.

Some researchers direct their work to the techniques that integrate cluster partitioning and instruction scheduling. Őzer *et al.* described a method called unified assign and

schedule (UAS), meant to take into account the resource constraints imposed by a partial schedule when a cluster assignment decision is made [49]. UAS employs list scheduler with a modification that reflects copy latency. Each instruction with available source operands is assigned to the cluster containing the highest priority. If no such cluster can be found, UAS will try to schedule the current instruction into the next cycle. The authors stated that UAS can utilize clusters more effectively and generate more compact schedules than BUG.

Sànchez *et al.* designed another unified technique that performs cluster assignment and instruction scheduling within one pass [54, 55]. The goal of their method is to find a minimal II for a given loop. Therefore, they allocate dependent instructions, especially those in a recurrence, in the same clusters as much as possible. In addition, because they observed fewer dependences crossing loop iterations than dependences within the same iteration in normal situations, they propose unrolling the loops and scheduling them when intercluster communication is a limiting factor for computing II.

Hiser *et al.* developed a global greedy algorithm for partitioning register resources on a clustered VLIW machine [32, 33]. They aimed to gain retargetability and flexibility in their code generator. For this reason their method, unlike other approaches that are based upon partitioning operation graphs, builds a data value graph whose nodes and edges are associated with machine dependent details. This makes it easier to generate code for special-purpose architectures such as DSP chips using an approach independent of scheduling and allocation algorithms. They applied their method to software pipelined loops and entire functions as well.

1.2.2 Loop Transformations

Loop transformations are important techniques for generating efficient code for a variety of architectures. This section summarizes research that employs loop transformations to enhance code performance for pipelined machines and shared-memory architectures as well as for improving memory performance.

1.2.2.1 Loop Fusion

Callahan, Cocke, and Kennedy proposed a metric based upon loop balance to measure efficiency of pipelined architectures [14]. They investigated the impact of loop

fusion on processor efficiency and found that the balance of a fused loop can be lowered by fusing a memory-bound loop and a compute-bound loop.

Allen, Callahan, and Kennedy developed a greedy algorithm for transforming sequential programs into equivalent parallel programs that can be run on a shared-memory machine [7]. Their method attempts to enhance the granularity of parallelism via loop fusion. This greedy algorithm is task parallelism oriented and may not achieve maximum loop-level parallelism. Later Kennedy and McKinley used a greedy fusion algorithm to maximize loop-level parallelism while minimizing the synchronization overhead [38]. They derived a fusion graph, where nodes represent loops and edges represent dependences between loops, from a program. Maximum parallelism granularity can be achieved by fusing parallel components of a fusion graph when semantic constraints imposed by the original program are satisfied.

These algorithms have only considered the cases when all loop headers are *conformable*, i.e., all loops have the same number of iterations and are all either parallel or sequential loops. Kennedy and McKinley improved their work by presenting a more general method called typed fusion [39]. This algorithm gives each loop a type based upon its header(s), and fuses the loops with the same type if the dependence constraints are not violated. The typed fusion algorithm takes $O((N+E)T)$ time, given a graph with N nodes, E edges, and T types.

Loop fusion for reuse was discussed in [9] where an ad hoc greedy algorithm was used without presenting a time bound. Later Kennedy and McKinley developed a weighted greedy fusion algorithm using weights of edges to represent the amount of data reuse and proved that the fusion problem for maximizing data locality is NP-hard [38]. In that paper, they also proposed to characterize the data reuse problem as a maximum-flow/minimum-cut problem that breaks the fusion-preventing edges to generate a minimum number of groups such that loops in each group can be fused together to achieve the best reuse. The complexity of the resulting algorithm is $O(kEN \log(N^2/E)$ where k is the application times of the maximum-flow algorithm.

Similar work was done by Gao *et al.* for improving reuse on uniprocessors [29]. Their work, which is also based on the maximum-flow/minimum-cut algorithm, takes a pre-processing step to pre-assign some nodes to partitions based on fusion-preventing edges. By doing so, they claim a good payoff in the running time can be obtained due to the reduced size of the input graph to the maxflow algorithm.

McKinley, Carr, and Tseng designed a loop cost model for their loop transformation framework to improve data locality [47]. Their algorithm groups the array references in a given loop nest based upon their data reuse types, and computes the number of cache line accesses using the cost model. The loop cost is used to guide loop optimizations including loop fusion. Loop fusion is profitable if the fused loop has a lower cost than the sum of the costs of the individual loops.

Kennedy proposed the first global fusion algorithm in [37]. This algorithm examines the entire program to determine collections of loops that can be fused together to obtain optimal reuse. The algorithm takes O(N(E+N))time and is faster than the straightforward greedy algorithm whose complexity is $O(E(E+N))$, when considering in practice $N \ll E$.

Although loop fusion has been studied extensively for improving both parallelism and reuse, how to apply this transformation for architectures with multiple register files is still unknown.

1.2.2.2 Scalar Replacement

Callahan, Cocke, and Kennedy proposed to eliminate redundant memory accesses via scalar replacement for improving efficiency of pipelined architectures [14]. Callahan, Carr, and Kennedy developed a general procedure for scalar replacement [13]. The algorithm is based on a pruned dependence graph, in which each true or input dependence represents an opportunity of an array element reuse. For each dependence $(source, sink)$ in the graph, $r + 1$ temporaries are used to hold the variables, where r is the number of loop iterations between *source* and *sink*. The algorithm then inserts the necessary register-to-register moves and initialization operations to keep the values of the temporaries correct. This work was extended by Carr *et al.* to address scalar replacement in the presence of inner-loop conditional control flow [20]. When computations involving subscripted variables are available only on certain paths, scalar replacement is incorporated with partial redundancy elimination to make computations available along each path. They reported integer-factor improvement over code generated by a commercial compiler of conventional design.

Duesterwalk, Gupta, and Soffa developed a data flow framework for array references that can support scalar replacement [24]. They modeled the flow of array references in a loop that may contain conditionals and determined the iteration distance value between a definition of an array reference and its use. This information can be used to assign

subscripted variables to registers. Their work differs from [20] in that it does not perform partial redundancy elimination.

1.2.2.3 Loop Unrolling and Unroll-and-jam

Loop unrolling has been shown an important technique for software pipelining to generate high performance code [45, 46, 52]. Unrolling enables more instructions from different iterations to be scheduled in the same cycle, and hence improves the resource utilization.

The impact of loop unrolling and unroll-and-jam on loop balance was examined in [14], where a method for updating the dependence graph after unrolling was developed. loop unrolling has no effect on machine efficiency if costs associated with control flow and address computation can be ignored when compared with floating-point computation. On the other hand unroll-and-jam introduces more computation into an innermost loop body without a proportional increase in memory references, and hence is a useful transformation for improving loop balance. Unfortunately, performing unroll-and-jam automatically is not straightforward. Excessive unroll-and-jamming may degrade the performance due to large amount of memory spill caused by high register pressure. A number of studies have been proposed to address this problem.

Carr and Kennedy investigated in depth the combination of scalar replacement and unroll-and-jam for ILP architectures [21]. Their method applied unroll-and-jam automatically to a loop nest based upon dependence analysis. The objective is to balance memory accesses and floating-point instructions and to satisfy register constraints on a specific target architecture. The experiment showed that a speedup of nearly 3 was attainable on loop kernels. That work suggested that automatic techniques for loop transformations, such as unroll-and-jam, can be not only possible but also extremely effective.

Another method for determining unroll factors was described in [19] by Carr and Guan. That technique used a data reuse model and a linear algebra framework from [64] to compute loop balance and register pressure. As a result, less memory space is needed for storing a dependence graph, as compared to the dependence-based approach in [21].

Carr utilized unroll-and-jam for improving ILP in the context of cache performance [17]. To do this, he modified the formula of computing loop balance so as to include the effects of cache misses and software prefetching.

Sarkar's work on selection of unroll-and-jam amounts has concentrated on ILP and instruction cache constraints [56]. The cost model in his work reflects the cost caused by register spill and instruction-cache misses as well as the degree of ILP in an unroll-and-jammed loop. An approach similar to computing ResII and RecII in modulo scheduling was used to compute ILP benefit in his cost model. Sarkar also designed a code generation algorithm to produce compact code for unrolled loops such that the resulting code includes fewer remainder loops when the loop iteration count is not a multiply of the unroll amount. He reported that this method obtained an average speedup of 1.08 in execution time on seven benchmarks.

Carr *et al.* used unroll-and-jam to improve ILP available for software pipelining [18]. Unroll-and-jammed loop can utilize the hardware resources more efficiently, allowing a software pipelining algorithm to find a better schedule. Their experiment showed an average improvement of 43% in II.

Huang *et al.* described a loop transformation scheme to enhance ILP for software pipelined loops on partitioned register file architectures [34]. Their method uses unroll-and-jam to increase parallelism within a single cluster, then unrolls the loop level with the fewest alignment conflicts to improve parallelism across clusters. The transformed program, after scalar replacement is applied to further improve register usage, is used to generate the final code using the method described in [32]. In their experiments, the transformed loops obtain 27% and 20% II improvement over untransformed loops on a 8-wide machine with 2 clusters and 4 clusters respectively. Their work is based upon an ad hoc approach; therefore, it does not make use of parameters of a specific architectures to transform a specific loop.

1.2.2.4 Loop Alignment

In the automatic scheme developed by Allen *et al.* for exposing parallelism from sequential programs, loop alignment was employed to create parallel loops by making loop-carried dependences loop independent [7]. They designed a technique, termed *code replication*, to eliminate alignment conflicts in a loop. Code replication splits the source of an alignment conflict into different statements such that the dependences in the alignment conflict can be aligned separately. They proved that the combination of code replication, loop alignment, and statement reordering is sufficient to eliminate all alignment conflicts in a loop. However, to find a minimum amount of replication is a NP-hard problem.

Chapter 2

Loop Transformation Strategy

This chapter begins with a brief description of the compiler organization used in this work and then demonstrates the effects of high-level transformations on clustered VLIW architectures. Finally, an overview of the loop transformation strategy for clustered VLIW architectures, followed by a discussion of transformation overhead, is given.

2.1 Compiler Structure

Figure 2.1 shows the optimizations used in this work. Optimizations are performed at two levels: high-level transformations that restructure high-level language programs, and low-level optimizations that are applied to low-level intermediate code.

The high-level transformation phase focuses on loop optimizations that reorder the execution of loop iterations for maximizing parallelism and data locality while minimizing the communication overhead. These transformations are performed on a high-level intermediate representation, which preserves all semantic information of source code programs. Transformations at a high-level can exploit optimizing opportunities that can not be detected easily by low-level optimizations, and hence increase the abilities of low-level optimizers to generate excellent code. For example, array references are easy to detect, whereas in low-level intermediate code, they appear as sequences of address calculations. Detection of array references provides high-level transformations a way to capture potential parallelism and predict communication overhead for a given loop nest, such that the loop can be transformed to enhance ILP available to a low-level code generator while maintaining semantic constraints.

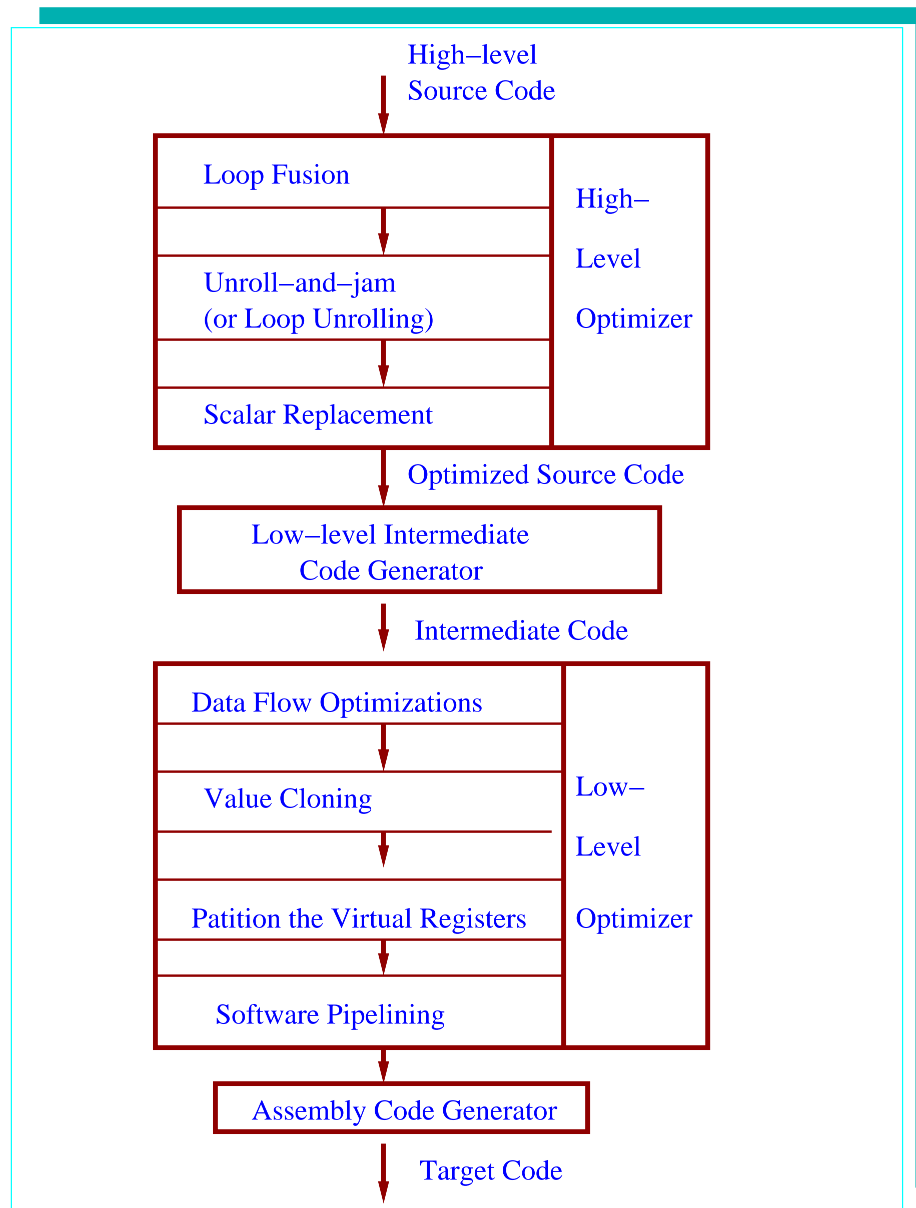

Figure 2.1: Compiler Structure

After high-level transformations, the transformed source code is converted into a low-level intermediate form. Low-level data flow optimizations, followed by register partitioning and software pipelining, are applied and, finally, the assembly code for the target machine is generated.

This research focuses on the high-level optimization phase. High-level transformations include loop fusion, loop unrolling /unroll-and-jam, and scalar replacement. The goal is to improve performance by maximizing concurrent use of functional units and minimizing the communication overhead among clusters. The transformed loops exploit considerable parallelism and data locality, and therefore are amenable to software pipelining to generate high performance code. In this work the initiation interval of software-pipelined loops is used to measure loop performance. The examples in the next section illustrate how loop fusion, loop unrolling, unroll-and-jam, and scalar replacement can be applied to improve II on a clustered VLIW machine.

2.2 Examples

By fusing two loops together, memory references are brought into the same loop body and hence are likely to be allocated into registers for reuse. In addition, loop fusion enables operations in the second loop to use the open instruction slots in the original loops, making more efficient use of functional units. Consequently fusion not only speeds up computation but also reduces power waste caused by idle functional units. The benefit of loop fusion on performance can be demonstrated by the following example. The original loops are listed below:

```
DO I = 1, 2*N
  s = 0
  DO J = 1, M
    s = s + H(J) * X(I,J)
  ENDDO
  A(I) = s
ENDDO
DO I = 1, 2*N
  t = 0
  DO J = 1, M
    t = t + L(J) * X(I,J)
  ENDDO
  B(I) = t
ENDDO
```

Suppose load and multiplication take two cycles to complete, and other operations take one cycle. Given a 4-functional unit 2-cluster machine, a schedule for the inner loop of the first loop needs three cycles as shown below (assume value cloning is applied [43]):

mult r1, h1, x1	nop	nop	nop
load h1,H(J)	load x1, X(I,J++)	nop	nop
add s, s, r1	nop	nop	nop

The second loop can be scheduled similarly. The two loops need IIs of six in total, leaving more than half of the functional units unused. Fusing the two loops results in a loop as follows

```
DO I = 1, 2*N
  s = 0
  t = 0
  DO J = 1, M
    s = s + H(J) * X(I,J)
    t = t + L(J) * X(I,J)
  ENDDO
  A(I) = s
  B(I) = t
ENDDO
```

and an II of 3, or a speedup of 2 over the original loops. One schedule of the fused loop is shown below:

mult r1, h1, x1	nop	mult r11, h,x	nop
load h1,H(J)	load x1, X(I,J++)	load h, H(J)	load x, X(I,J++)
add s, s, r1	nop	add t, t, r11	nop

The fused loop can be further transformed to improve usage of machine resources in a single cluster via unroll-and-jam and scalar replacement as described in Section 1.1.4. Unroll-and-jam can move outer loop-carried reuse into the innermost loop and improve software pipelining. Scalar replacement replaces memory references with temporary scalar variables, making them easier to be allocated to registers.

Furthermore, unroll-and-jam or loop unrolling can be performed to increase the data independent parallelism crossing different clusters. The resulting loop provides multiple copies of the loop body that can be executed in parallel on different clusters. This transformation is analogous to a parallel loop where different iterations run on different processors. On a clustered architecture, each unrolled iteration can be executed on a separate cluster. While on shared memory machines the startup of loop parallelism can cause significant overhead, in the context of clustered architectures there is no such overhead [34].

For example, unroll-and-jamming the fused loop by a factor of 2 produces the loop

```
   DO I = 1, 2*N, 2
     s1 = 0
     t1 = 0
     s2 = 0
     t2 = 0
     DO J = 1, M
C    iteration i
       s1 = s1 + H(J) * X(I,J)
       t1 = t1 + L(J) * X(I,J)
C    iteration i+1
       s2 = s2 + H(J) * X(I+1,J)
       t2 = t2 + L(J) * X(I+1,J)
     ENDDO
     A(I) = s1
     B(I) = t1
     A(I+1) = s2
     B(I+1) = t2
   ENDDO
```

Since there are no dependences between the iterations, no intercluster communication is required if iterations `i` and `i+1` are executed on different clusters.

Performing scalar replacement can remove redundant memory accesses, but may cause intercluster communication. After scalar replacement the above loop becomes

```
   DO I = 1, 2*N, 2
     s1 = 0
     t1 = 0
     s2 = 0
     t2 = 0
     DO J = 1, M
C    iteration i
       hh = H(J)
       ll = L(J)
       x1 = X(I,J)
       s1 = s1 + hh * x1
       t1 = t1 + ll * x1
C    iteration i+1
       x2 = X(I+1,J)
       s2 = s2 + hh * x2
       t2 = t2 + ll * x2
     ENDDO
     A(I) = s1
     B(I) = t1
     A(I+1) = s2
     B(I+1) = t2
   ENDDO
```

The additional copy operations are needed to transfer the values of `H(J)` and `L(J)` between clusters. The latencies for intercluster copies can sometimes be hidden in the software pipelined code. For example, one potential schedule for the above transformed loop is

mult r1, h1, x1	mult r2, l1, x1	mult r11, h2, x2	mult r12, l2, x2
load h1, H(J)	load l1, L(J)	load x2, X(I+1,J++)	nop
load x1, X(I,J++)	nop	add s2, s2, r11	add t2, t2, r12
add s1, s1, r1	add t1, t1, r2	copy h2, h1	copy l2, l1

which results in an II of 4 for the loop kernel. This means a unit II (II per unrolled iteration) of 2, or a speedup of 3 over the original loop.

Typically communication between clusters occurs when the inner loop carries a dependence. Consider the loop

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

and the unrolled loop

```
   DO I = 1, 2*N, 2
C    iteration i
     A(I) = A(I-1) + 1
C    iteration i+1
     A(I+1) = A(I) + 1
   ENDDO
```

Before unrolling there is a true dependence from the statement to itself carried by the `I`-loop. After unrolling by a factor of 2, there are two dependences between the statements. If the code for iteration `i` is executed on one cluster and the code for iteration `i+1` on another, the loop schedule requires communication between clusters.

Not all loop-carried dependences require communication. As in shared-memory parallel code generation, loop alignment can be used to change a loop-carried dependence into a loop-independent dependence such that the aligned loop carries no dependences and is amenable to loop unrolling or unroll-and-jam. However, when alignment conflicts exist (as in the previous example), intercluster communication becomes inevitable if unrolled loop iterations are distributed in distinct clusters. Actually aligning a loop is unnecessary to expose the intercluster parallelism as [34] demonstrates. Consider the following loop:

```
   DO I = 1, 2*N
     A(I) = B(I) + Q
     C(I) = A(I-1) + X(I)
   ENDDO
```

Unrolling this loop once for a 2-cluster machine produces:

```
   DO I = 1, 2*N, 2
     A(I) = B(I) + Q
     C(I) = A(I-1) + X(I)
     A(I+1) = B(I+1) + Q
     C(I+1) = A(I) + X(I+1)
   ENDDO
```

The low-level optimizer can keep the source and sink of a dependence on the same cluster by assigning the first and the forth statements into one cluster, and the other two statements into another cluster. This would result in the same effect as loop alignment since no

communication is needed. Therefore, loop alignment is not applied in this work. Instead the low-level optimizer is responsible for capturing parallelism in the unrolled loop with alignable dependences. In fact, alignment conflicts are used in this work to determine the intercluster communication cost in a loop.

2.3 Loop Transformation Strategy

When various transformations are performed, the benefit obtained by one individual transformation may be negated by other transformations. Careless organization of transformations can degrade collective performance. This gives rise to an ordering issue. The complete loop transformation strategy is listed below; namely, loop transformations are performed in the following order:

1. Fuse loops to enhance intracluster parallelism and, later, exploit intercluster parallelism by unrolling.

2. Unroll-and-jam (or unroll) loops to increase intracluster parallelism and enhance data-independent parallelism across multiple clusters.

3. Perform scalar replacement to improve register usage.

Loop fusion and unrolling will increase the size of the loop body. While enlarged code scope provides opportunities for optimizations, an excessively large loop body may make it difficult for software pipelining to find an efficient schedule. High register pressure that results from a large loop body also hampers the overall performance. The transformation strategy in this research performs loop fusion before unroll-and-jam (or loop unrolling) in order to avoid unprofitable loop body size, because when unroll-and-jam is performed, the code size and register pressure can be limited by restricting unroll amounts. Both loop fusion and unroll-and-jam provide opportunities for improving data locality via scalar replacement; therefore, it is reasonable to perform scalar replacement during the last step.

2.4 Overhead

When the trip count of a loop (i.e., the number of times a loop body is executed) is unknown, or is not a multiple of the unroll amount, a few loop iterations must be moved

from the beginning or end of the loop and executed separately such that loop unrolling or unroll-and-jam is applicable. This transformation is called *loop peeling*. Loop peeling is useful for applying loop fusion as well. When two loops cannot be fused together due to different trip counts, peeling the loop with a larger trip count enables loop fusion. The code for the peeled iterations can be enclosed within a conditional or a new loop, introducing an extra control overhead. When the loop to be transformed is iterated a small number of times, the overhead caused by loop peeling may overcome the benefit of loop unrolling (or fusion) and degrade the overall performance. The experiments of this research showed that the overhead of peeling loops is ignorable if the trip count of a loop is greater than 32.

Chapter 3

Unroll-and-jam

To generate a good software pipeline for unroll-and-jammed loops on clustered VLIW machines, a compiler should not only be able to determine which dependences introduce intercluster communication, but also predict whether the communication increases the schedule length of the software pipelined code. This needs to be done before any unroll-and-jam is performed. The method in this section uses unit MinII, or *uMinII*, including the effects of intercluster data transfers, as a metric to guide unroll-and-jam for clustered VLIW machines. It performs unroll-and-jam on the loop levels that will create the most intracluster and/or intercluster parallelism and determines the unroll-and-jam amounts that will give the best loop performance.

This chapter is organized as follows. Section 3.1 discusses the safety of unroll-and-jam and a method for updating the dependence graph for unroll-and-jammed loops. Section 3.2 provides a heuristic approach for picking loops to unroll. Section 3.3 describes a method to compute *uMinII* for unroll-and-jammed loops. Section 3.4 gives the formula for register pressure caused by unroll-and-jam. Finally, Section 3.5 presents a method for computing unroll-and-jam amounts to achieve a lower *uMinII*.

3.1 Safety and Updating the Dependence Graph

Unroll-and-jam changes the execution order of loop iterations. Consider the following loop,

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

Unroll-and-jamming the `I` loop once results in

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

Note that the original loop executes the instance of the statement for the kth iteration of the `I`-loop after all instances for the $(k-1)^{th}$ iteration of the `I`-loop are completed. In the transformed loop, however, two iterations of the `I`-loop are executed on the same iteration of the `J`-loop. In this example, unroll-and-jam reverses the source and sink for some dependences. For instance, the original loop stores into the location of `A(2,2)` on iteration $\langle 1,2\rangle$, and reads from the same location on iteration $\langle 2,1\rangle$. After unroll-and-jam, the read of `A(2,2)` is brought prior to its definition, as the value is accessed on iteration $\langle 1,1\rangle$ and is assigned one iteration of the `J`-loop later, i.e., when `I=1`, `J=2`. This is illegal, since the access order of the same memory location is changed by unroll-and-jam. The data dependences reflect this changes. Unroll-and-jam converts a true dependence with a distance vector of $\langle 1,-1\rangle$ into two dependences: an anti dependence $\langle 0,1\rangle$ and a true dependence $\langle 1,-1\rangle$. The new anti dependence indicates that a reverse of execution order occurs.

The following example discusses the constraints of dependences on unroll amounts in detail. In the loop

```
DO I = 1, 6*N
  DO J = 1, N
    A(I+2,J) = A(I,J+1) + B(I,J)
  ENDO
ENDDO
```

a true dependence $\langle 2,-1\rangle$ exists between `A(I+2,J)` and `A(I,J+1)`. Unroll-and-jamming the `I`-loop once results in

```
DO I = 1, 6*N, 2
  DO J = 1, N
    A(I+2,J) = A(I,J+1) + B(I,J)
    A(I+3,J) = A(I+1,J+1) + B(I+1,J)
  ENDO
ENDDO
```

where two true dependences with the distance vector $\langle 1, -1 \rangle$ guarantee that the sources of dependences are executed before the sinks. However unrolling the outer loop by a factor of three would reverse the dependence, as shown below

```
DO I = 1, 6*N, 3
  DO J = 1, N
    A(I+2,J) = A(I,J+1) + B(I,J)
    A(I+3,J) = A(I+1,J+1) + B(I+1,J)
    A(I+4,J) = A(I+2,J+1) + B(I+2,J)
  ENDO
ENDDO
```

An anti dependence $\langle 0, 1 \rangle$ goes from `A(I+2,J+1)` to `A(I+2,J)`, reversing the memory access order. This means unroll-and-jamming by a factor of three is illegal for this example.

Callahan *et al.* proved the following theorem which lists the conditions under which unroll-and-jam is legal [14]:

Theorem 1 *An unroll-and-jam factor n is legal if and only if there exists no dependence with direction vector $(<, >)$ such that the distance for the outer loop is $< n$.*

When unroll-and-jam is legal, the data dependences for the unroll-and-jammed loop are used to examine the effect of unroll-and-jam on intercluster communication. Callahan *et al.* proposed a method for computing the data dependence graph (DDG) after loop unrolling for consistent dependences whose array references contain only one loop induction variable in each subscript position and whose references are variant with respect to the unrolled loop [14].

If we unroll the mth loop in a nest L k times, then the updated dependence graph $G' = (V', E')$ for the unrolled loop L' can be computed from the dependence graph $G = (V, E)$ for L by the following rules:

1. For each $v \in V$, there are $k + 1$ nodes $v_0, \ldots, v_k$ in V'. These correspond to the original reference and its k copies.

2. For each edge $e = \langle v, w \rangle \in E$, with distance vector $d(e)$, let $d_m(e)$ be the mth entry of $d(e)$, there are $k+1$ edges $e_0, \ldots, e_k$ where $e_j = \langle v_j, w_i \rangle$, v_j is the jth copy of v, w_i is the ith copy of w and

$$
\begin{aligned}
i &= (j + d_m(e)) \bmod (k+1) \\
d_m(e_j) &= \begin{cases} \lfloor \frac{d_m(e)}{k+1} \rfloor & \text{if } i \geq j \\ \lfloor \frac{d_m(e)}{k+1} \rfloor + 1 & \text{if } i < j \ . \end{cases}
\end{aligned}
$$

Consider, as an example, the following loop

```
DO I = 1, 3*N
  A(I) = A(I-1) + Q
ENDDO
```

where a dependence with distance vector $\langle 1 \rangle$ exits. Unrolling this loop twice produces

```
DO I = 1, 3*N, 3
  A(I) = A(I-1) + Q
  A(I+1) = A(I) + Q
  A(I+2) = A(I+1) + Q
ENDDO
```

The updated dependence graph contains three nodes, v_0, v_1, and v_2, representing three statements in the unrolled loop. Two forward dependence edges, (v_0, v_1) and (v_1, v_2), have a distance of 0 ($\lfloor 1/3 \rfloor = 0$), and one backward edge, (v_2, v_0), has a distance of 1 ($\lfloor 1/3 \rfloor + 1 = 1$), as shown in Figure 3.1.

In this research, the updated dependence graph is utilized for determining the dependences in the innermost loop after loop unrolling or unroll-and-jam is applied.

3.2 Determining Loops to Unroll

In this research unroll-and-jam is applied to achieve both intracluster and intercluster parallelism. The first step of this approach is to determine the loop levels for unrolling or unroll-and-jam. For improving ILP in a single cluster, this method seeks the loop level, l_a, that will have the most data reuse after unroll-and-jam is applied. In other words, this method chooses the loop level that carries the most dependences that can become amenable to scalar replacement after applying unroll-and-jam. To obtain intercluster parallelism, we

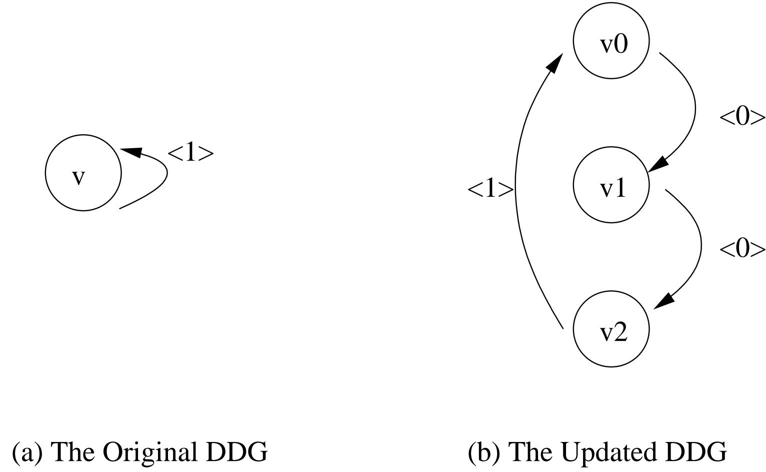

Figure 3.1: Updating the Dependence Graph

unroll the loop level, l_p, that contains the fewest dependences that may result in intercluster data transfers after unrolling.

To aid in the discussion that follows we define U_a as the unroll factor for l_a and U_p as the unroll factor for l_p. Hence, the unroll factor of the entire loop nest is $U_a \times U_p$. Note that l_a and l_p can be the same loop.

3.3 Computing *unit MinII*

To compute the unit MinII (*uMinII*) of a loop, both the unit *RecII* (*uRecII*) and the unit *ResII* (*uResII*) need to be computed. Callahan *et al.* have shown that unrolling and unroll-and-jam do not increase the *uRecII* of a loop [14]. Thus, this method only computes the *uRecII* once. If the innermost loop is unrolled, then the *uRecII* remains unchanged. If an outer loop is unrolled, then the *uRecII* decreases by a factor of the unroll amount.

On an architecture with hardware loop support, FU_f fixed-point units, FU_m memory/address units and support for FU_c intercluster copies per cycle, the *uResII* is

$$\frac{\max\{\lceil \frac{F}{FU_f} \rceil, \lceil \frac{M}{FU_m} \rceil, \lceil \frac{C}{FU_c} \rceil\}}{U_a \times U_p}.$$

Here F is the number of fixed-point operations in the loop body, M is the number of memory references and C is the number of intercluster data transfers. F is defined as

$$f \times \prod_{1 \le i \le n} U_i,$$

where f is the original number of fixed-point operations in the loop, U_i is the unroll amount of the ith loop, and n is the loop nesting depth. $M = M_L \times U_n$, where M_L is the number of memory references in the loop after unroll-and-jam and scalar replacement, and U_n is the unroll amount of the innermost loop. The computation of M_L is outlined in detail by Carr and Kennedy [21].

When computing C it is assumed that the unroll-and-jammed loop is partitioned in such a way that U_a copies of loop body derived from unroll-and-jamming loop l_a are placed in a single cluster. Then each of the U_p copies of this statement group is executed in distinct clusters. There are two reasons for this assumption. First, by distributing the copies of the same statement group in separate clusters, a balanced work load across clusters is likely obtained. Second, this partitioning scheme keeps many operations involving data reuse within a single cluster, which limits intercluster communication.

Section 3.3.1 addresses computing $uMinII$ when a single loop is unrolled. Then the discussion is extended to unrolling multiple loops in Section 3.3.2.

3.3.1 Unrolling a Single Loop

When computing the intercluster copies caused by unrolling a single loop l ($l = l_a = l_p$), we consider the dependence graph $G_l = (V_l, E_l)$ consisting of all dependences where the distance vector associated with the dependence, $d(e)$, is of the form $\langle 0, \ldots, 0, d_l, 0, \ldots, d_n \rangle$ such that the lth entry of $d(e)$, $d_l(e)$, is not 0. E_l is partitioned into two groups: E_l^C and E_l^I. E_l^C is the set of unalignable dependences carried by l whose source and sink are variant with respect to l. E_l^I is the set of dependences whose references are invariant with respect to l. The communication cost due to edges in E_l^C is denoted C_l^C and the communication cost due to edges in E_l^I is denoted C_l^I. For simplicity of presentation, it is assumed that each array reference has at most one incoming dependence edge. If a reference has multiple incoming edges, a distance vector is constructed such that its ith entry has the minimum value over all incoming edges. This vector represents the possible earliest point in unrolling from where the reference can get its value. See [21] for details on handling multiple incoming edges.

Computing C_l^C: For each edge $e = (v_0, w_0) \in E_l$ there are $U_p \times U_a$ edges e_0, e_1,...,$e_{U_p \times U_a - 1}$ $\in E_l'$ after unroll-and-jam. For each $e_m = (v_m, w_n) \in E_l'$, $n = (m + d_l(e)) \bmod (U_p \times U_a)$ [14]. We examine the first U_a edges created by unrolling to determine the communication cost per cluster for l, denoted $uC_l(e)$ for each edge $e \in E_l$. Since the sources of the first U_a edges, $v_0, v_1, \ldots, v_{U_a - 1}$, will be located in the first cluster, communication exists if and only if any sink is not in the first cluster, i.e., if any $n \geq U_a$. This implies that $uC_l(e)$ is the number of edges where $n = (m + d_l(e)) \bmod (U_p \times U_a) \geq U_a$, and $m = 0, 1, \ldots, U_a - 1$. Hence,

$$C_l^C = \sum_{e \in E_l^C} uC_l(e) \times U_p.$$

To derive $uC_l(e)$, we break down the computation into four cases. These cases arise from the fact that dependences in loops are usually very simple. For each case we give our conclusion and proof followed by an example.

Case 1: If $d_l(e) \bmod (U_a \times U_p) = 0$, then $uC_l(e) = 0$.

Proof:

$$n = (m + d_l(e)) \bmod (U_p \times U_a) = (m + d_l(e)) - \lfloor \frac{m + d_l(e)}{U_p \times U_a} \rfloor U_p \times U_a.$$

If $d_l(e) \bmod (U_a \times U_p) = 0$, then there exists a $t = 0, 1, 2, \ldots$, such that $d_l(e) = t(U_a \times U_p)$. This gives

$$m + d_l(e) = m + t(U_p \times U_a).$$

Since $0 \leq m < U_a$,

$$tU_p \times U_a < m + t(U_p \times U_a) < (tU_p + 1)U_a.$$

From this follows

$$\begin{aligned} \tfrac{tU_p \times U_a}{U_p \times U_a} &< \tfrac{m + d_l(e)}{U_p \times U_a} \\ &< \tfrac{(tU_p + 1)U_a}{U_p \times U_a} \\ &< \tfrac{(tU_p + U_p)U_a}{U_p \times U_a} \quad (since\ U_p \geq 2). \end{aligned}$$

Thus,

$$t < \frac{m + d_l(e)}{U_p \times U_a} < t + 1.$$

So

$$\lfloor \frac{m + d_l(e)}{U_p \times U_a} \rfloor = t.$$

This gives

$$\begin{aligned} n = & \quad (m + d_l(e)) - \lfloor \tfrac{m+d_l(e)}{U_p \times U_a} \rfloor U_p \times U_a \\ = & \quad m + d_l(e) - t(U_p \times U_a) \\ = & \quad m\ . \end{aligned}$$

Since $m < U_a$, then $n < U_a$, which means there exists no communication, i.e., $uC_l(e) = 0$.

Example:

Consider the following loop where $d_l(e) = 4$:

```
DO I = 1, 4*N
  A(I) = A(I-4)
ENDDO
```

To generate code for a 4-cluster machine, the loop is unrolled by a factor of 4, giving

```
DO I = 1, 4*N, 4
  A(I)   = A(I-4)
  A(I+1) = A(I-3)
  A(I+2) = A(I-2)
  A(I+3) = A(I-1)
ENDDO
```

Each statement may be executed on a different cluster without incurring a communication cost due to the original dependence.

Case 2: If $U_a = 1$ and $d_l(e)$ mod $(U_a \times U_p) \neq 0$, then $uC_l(e) = 1$.

Proof:

Since $U_a = 1$, each cluster has one copy of loop body, i.e., $m = 0$.

$$\begin{aligned} n = & \quad (m + d_l(e)) - \lfloor \tfrac{m+d_l(e)}{U_p \times U_a} \rfloor U_p \times U_a \\ = & \quad d_l(e) - \lfloor \tfrac{d_l(e)}{U_p} \rfloor U_p. \end{aligned}$$

If $n = 0$, $d_l(e) = \lfloor \frac{d_l(e)}{U_p} \rfloor U_p$. Then $d_l(e)$ mod $(U_a \times U_p) = 0$. This is a contradiction. Therefore $n \neq 0$, giving $uC_l(e) = 1$.

Example:

Consider the following loop where $d_l(e) = 1$:

```
DO I = 1, 3*N
  A(I) = A(I-1)
ENDDO
```

After unrolling by a factor of 3, where $U_p = 3, U_a = 1$, we have

```
DO I = 1, 3*N, 3
  A(I)   = A(I-1)
  A(I+1) = A(I)
  A(I+2) = A(I+1)
ENDDO
```

If each of these statements is assigned to a different cluster, each cluster needs one intercluster data transfer.

Case 3: If $d_l(e) \bmod (U_a \times U_p) \neq 0$, $U_a > 1$, and $U_a \geq d_l(e)$, then $uC_l(e) = d_l(e)$.

Proof:

Since $m < U_a$, and $d_l(e) \leq U_a$, then $m + d_l(e) < 2U_a$.

$U_p \geq 2$, so $m + d_l(e) < U_p \times U_a$, or

$$\lfloor \frac{m + d_l(e)}{U_p \times U_a} \rfloor = 0.$$

From this follows

$$\begin{aligned} n = & \ (m + d_l(e)) \bmod (U_p \times U_a) \\ = & \ m + d_l(e) - \lfloor \tfrac{m+d_l(e)}{U_p \times U_a} \rfloor U_p \times U_a \\ = & \ m + d_l(e). \end{aligned}$$

The value of $uC_l(e)$ equals the number of m whose value makes $n \geq U_a$ hold. When $m \geq U_a - d_l(e)$, $n \geq U_a$. Among $m = 0,\ 1,\ ....,\ U_a - d_l(e),\ U_a - d_l(e) + 1,\ ...,\ U_a - 1$ there are $d_l(e)$ $m's$ whose values are greater than or equal to $U_a - d_l(e)$. Therefore, $uC_l(e) = d_l(e)$.

Example:

Consider the following loop where $d_l(e) = 2$:

```
DO I = 1, 6*N
  A(I) = A(I-2)
ENDDO
```

After unrolling by a factor of 6 (assume $U_p = 2, U_a = 3$), the loop becomes

```
DO I = 1, 6*N, 6
  A(I)   = A(I-2)
  A(I+1) = A(I-1)
  A(I+2) = A(I)
  A(I+3) = A(I+1)
  A(I+4) = A(I+2)
  A(I+5) = A(I+3)
ENDDO
```

If the first three statements are executed on one cluster and the last three on another cluster, each cluster needs two intercluster copies.

Case 4: If $1 < U_a < d_l(e) < (U_a \times U_p)$, and $d_l(e) \bmod (U_a \times U_p) \neq 0$ then $uC_l(e) = \min(U_p \times U_a - d_l(e), U_a)$.

Proof:

We consider two cases:

case 1: $m < U_a \times U_p - d_l(e)$, or $m + d_l(e) < U_a \times U_p$. This gives

$$\lfloor \frac{m + d_l(e)}{U_p \times U_a} \rfloor = 0.$$

Then

$$\begin{aligned} n = & \quad (m + d_l(e)) \bmod (U_p \times U_a) \\ = & \quad m + d_l(e) - \lfloor \tfrac{m+d_l(e)}{U_p \times U_a} \rfloor U_p \times U_a \\ = & \quad m + d_l(e) \\ > & \quad U_a \qquad\qquad (\textit{since } d_l(e) > U_a \textit{ is assumed}). \end{aligned}$$

This means when $m < U_p \times U_a - d_l(e)$, $n > U_a$. If $U_p \times U_a - d_l(e) < U_a$, each cluster has $U_p \times U_a - d_l(e)$ value copies. Otherwise each cluster needs U_a value copies.

case 2: $m \geq U_p \times U_a - d_l(e)$, or $m + d_l(e) \geq U_p \times U_a$.

Since $m < U_a$, and $d_l(e) < U_p \times U_a$ (the assumption),

$$U_p \times U_a \leq m + d_l(e) < (U_p + 1) \times U_a$$

Then

$$\begin{aligned} 1 &\leq \tfrac{m+d_l(e)}{U_p \times U_a} \\ &< \tfrac{U_p+1}{U_p} \\ &< \tfrac{U_p+U_p}{U_p} \quad (since\ U_p \geq 2) \\ &< 2. \end{aligned}$$

From this follows

$$\lfloor \frac{m + d_l(e)}{U_p \times U_a} \rfloor = 1,$$

and

$$\begin{aligned} n = &\quad m + d_l(e) - \lfloor \tfrac{m+d_l(e)}{U_p \times U_a} \rfloor U_p \times U_a \\ = &\quad m + d_l(e) - U_p \times U_a. \end{aligned}$$

Since $d_l(e) < U_p \times U_a$ (the assumption), i.e., $d_l(e) - U_p \times U_a < 0$, and $m < U_a$, we have $m + d_l(e) - U_p \times U_a < U_a$. This shows that when $m \geq U_p \times U_a - d_l(e)$ no communication is needed.

Combining the two cases gives $uC_l(e) = \min(U_p \times U_a - d_l(e), U_a)$.

Example:

The original loop

```
    DO I = 1, 6*N
       A(I) = A(I-3)
    ENDDO
```

has $d_l(e) = 3$. Unrolling by a factor of 6 ($U_p = 3, U_a = 2$) results in a new loop as follows

```
    DO I = 1, 6*N, 6
S0:   A(I)   = A(I-3)
S1:   A(I+1) = A(I-2)
S2:   A(I+2) = A(I-1)
S3:   A(I+3) = A(I)
S4:   A(I+4) = A(I+1)
S5:   A(I+5) = A(I+2)
    ENDDO
```

If we partition the new loop as $\{S_0, S_1\}$, $\{S_2, S_3\}$, and $\{S_4, S_5\}$, each cluster will need 2 ($\min(3 \times 2 - 3, 2) = 2$) value copies.

Here is another example where $d_l(e) = 5$:

```
DO I = 1, 8*N
   A(I) = A(I-5)
ENDDO
```

Unrolling this loop 7 times ($U_p = 2, U_a = 4$) gives

```
DO I = 1, 8*N, 8
   A(I)   = A(I-5)
   A(I+1) = A(I-4)
   A(I+2) = A(I-3)
   A(I+3) = A(I-2)
   A(I+4) = A(I-1)
   A(I+5) = A(I)
   A(I+6) = A(I+1)
   A(I+7) = A(I+2)
ENDDO
```

If the first four statements are placed in one cluster, and the last four statements are placed in another cluster, each cluster needs 3 ($\min(2 \times 4 - 5, 4) = 3$) value copies.

In practice most dependences fall into the first three cases. From the derivation of $uC_l(e)$ for those three cases, the following observation can be derived: when unrolling a single loop l, the communication cost per cluster caused by dependences that are variant with respect to l does not change much with respect to the unroll factor. To compute $uC_l(e)$ for variant references in general, the code in Figure 3.2 can be used.

Computing C_l^I: For each dependence whose source and sink are invariant with respect to l, there are $U_a \times U_p$ references in the loop body after unroll-and-jam/unrolling. $U_a \times U_p - 1$ memory references are eliminated by scalar replacement. When partitioning the loop body into U_p separate clusters, a memory operation is executed in one cluster and the other $U_p - 1$ clusters require a copy operation. This function remains constant as U_a changes, giving

$$C_l^I = \sum_{e \in E_l^I} (U_p - 1).$$

3.3.2 Unrolling Multiple Loops

When $l_a \neq l_p$, we must consider the effects of multiple loops. Unroll-and-jam of l_a will potentially increase the number of loop-carried dependences causing communication

```
Procedure CommCost(d_l(e), U_p, U_a)
Input: d_l(e): dependence distance
       U_p, U_a: unroll amounts
Output: uC_l(e): the unit communication cost

if d_l(e) mod (U_a × U_p) = 0
  uC_l(e)=0
else if U_a = 1
  uC_l(e) = 1
else if U_a ≥ d_l(e)
  uC_l(e) = d_l(e)
else if (U_p × U_a) > d_l(e)
  uC_l(e)=min (U_p × U_a − d_l(e), U_a)
else
  uC_l(e)=0
  for ( m = 0; m < U_a; m++)
    if ( (m + d_l(e)) mod (U_p × U_a) > U_a)
      uC_l(e)=uC_l(e)+1
```

Figure 3.2: Compute Unit Communication Cost for E_l^C

when l_p is unrolled or unroll-and-jammed to spread computation across clusters. For references invariant with respect to l_p, the communication costs can be computed as described in the previous section. However, the same is not true for variant references.

To compute the communication cost for variant references, we consider the case when l_p is the innermost loop and the case when it is not. When l_p is the innermost loop, the two types of dependences that can cause communication are

1. innermost-loop-carried dependences, and
2. outer-loop-carried dependences that become carried by the innermost loop after unroll-and-jam.

In the first case, the method presented in the previous section accurately computes the communication cost. Note that when computing $uC_n(e)$ for this case, U_a and U_p are 1 and U_n respectively. Therefore, either case 1 or case 2 described in the previous section is used.

For the second case, only dependences with a distance vector of the form $\langle 0,\ldots,d_i, \ldots,0,\ldots,d_n\rangle$, where $d_i \neq 0$ and i is l_a, are considered. This set of dependences is denoted E_i^U. Consider any $e \in E_i^U$, after performing unroll-and-jam on i by a factor of U_i, $(U_i - d_i(e))^+$ dependences have a zero entry at i^{th} component of the distance vector and are carried by the innermost loop or are loop independent[1] [21].

For each edge $e \in E_i^U$ made innermost by unroll-and-jam, we need to compute the communication cost caused by unrolling l_p (the innermost loop). To derive this communication cost, we use the convention that $C_n(e)$, where n is l_p, is the communication cost caused by the dependences with distance $d_n(e)$ after unrolling the innermost loop $U_n - 1$ times. Since unrolling the innermost loop cannot increase data reuse, the only reason for unrolling the innermost loop is to create intercluster parallelism.

From the previous section, we have

$$C_n(e) = uC_n(e) \times U_n.$$

Thus, if the set of unalignable innermost-loop-carried dependences is denoted E_n^C, then the communication cost when l_p is the innermost loop is

[1] x^+ is defined as:

$$x^+ = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

$$\sum_{e \in E_n^C} U_i C_n(e) + \sum_{e \in E_i^U} (U_i - d_i(e))^+ C_n(e).$$

When l_p is not the innermost loop, we also only consider the dependences that can be made innermost. This set of dependences is denoted E_{ij}^U and the distance vector of each of these edges is of the form $\langle 0, \ldots, d_i, \ldots, 0, d_j, 0, \ldots, d_n \rangle$, where i is l_a and j is l_p, or vice versa. In this case both i and j must be unroll-and-jammed enough to make $d_i = 0$ and $d_j = 0$, giving a communication cost of

$$\sum_{e \in E_{ij}^U} (U_i - d_i(e))^+ (U_j - d_j(e))^+.$$

Note that $C_n(e)$ is not needed here since the innermost loop is not unrolled.

Example To demonstrate intercluster copy prediction for unrolling multiple loops, consider the following loop:

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

Let e_1 denote the edge from `A(I,J)` to `A(I-1,J)`, e_2 denote the edge from `A(I,J)` to `A(I,J-1)`, and e_3 denote the edge from `A(I,J)` to `A(I-1,J-1)`. Note that $d(e_1) = \langle 0, 1 \rangle$, $d(e_2) = \langle 1, 0 \rangle$ and $d(e_3) = \langle 1, 1 \rangle$. If $U_i = 2$, $U_j = 2$ for a 2-cluster machine, we have

$$\begin{aligned}
E_n^C &= \{e_1\} \\
E_j^U &= \{e_2, e_3\} \\
C_n(e_1) &= 1 \times 2 = 2 \\
C_n(e_2) &= 0 \\
C_n(e_3) &= 1 \times 2 = 2
\end{aligned}$$

Therefore,

$$
\begin{aligned}
C = \;& U_j \times C_n(e_1) + (U_j - d_j(e_2))^+ \times C_n(e_2) + \\
& (U_j - d_j(e_3))^+ \times C_n(e_3) \\
= \;& 2 \times 2 + 1 \times 0 + 1 \times 2 \\
= \;& 6
\end{aligned}
$$

3.4 Register Pressure

Unroll-and-jam/unrolling can increase the number of registers needed in the innermost loop body. Carr and Kennedy have presented a method to compute the number of registers required by scalar replacement for an unroll-and-jammed loop before unroll-and-jam is applied [21]. Their work does not consider unrolling an innermost loop, and is extended in this section to include the effects of inner-loop unrolling.

In computing register pressure R, the method proposed in [21] partitions the reference set of a loop into the following three sets that exhibit different memory behavior when unroll-and-jam is applied.

- $V^{\emptyset}$ is the set of references without an incoming dependence,
- V_r^C is the set of memory reads that have a loop-carried or loop-independent incoming dependence, but are not invariant with respect to any loop, and
- V_r^I is the set of memory reads that are invariant with respect to a loop.

The number of registers required by each reference set is represented by $R^{\emptyset}$, R_r^C, and R_r^I respectively, giving $R = R^{\emptyset} + R_r^C + R_r^I$. We have

(1) $R^{\emptyset} = 0$ since the references in $V^{\emptyset}$ are not amenable to scalar replacement.

(2) For each reference $v \in V_r^C$ such that the edge associated with v is carried by the innermost loop n, unrolling the innermost loop by U_n will create U_n dependence edges amenable to scalar replacement. From Callahan *et al.*, $(U_n - d_n(e_v))^+$ of these edges have distances of $\lfloor d_n(e_v)/U_n \rfloor$, and $\min(U_n, d_n(e_v))$ of them have distances of $\lfloor d_n(e_v)/U_n \rfloor + 1$ [14]. Therefore, for each dependence in the innermost loop, the number of registers required for scalar replacement after unrolling the innermost loop is

$$
\begin{aligned}
R_n \;=\;& (U_n - d_n(e_v))^+ \times \left(\left\lfloor \frac{d_n(e_v)}{U_n} \right\rfloor + 1 \right) + \\
& \min(U_n, d_n(e_v)) \times \left(\left\lfloor \frac{d_n(e_v)}{U_n} \right\rfloor + 2 \right).
\end{aligned}
$$

Since Carr and Kennedy show that

$$R_r^C = \sum_{v \in V_r^C} (\prod_{1 \le i < n} (U_i - d_i(e_v))^+) \times (d_n(e_v) + 1)$$

when the innermost loop is not unrolled, we have

$$R_r^C = \sum_{v \in V_r^C} \prod_{1 \le i < n} (U_i - d_i(e_v))^+ R_n$$

when the innermost loop is unrolled.
If $d_n(e_v) \ge U_n$, then $(U_n - d_n(e_v))^+ = 0$. This gives

$$R_n = U_n \times (\lfloor \frac{d_n(e_v)}{U_n} \rfloor + 2).$$

If $d_n(e_v) < U_n$, then

$$\left\lfloor \frac{d_n(e_v)}{U_n} \right\rfloor = 0.$$

Thus, R_n becomes

$$(U_n - d_n(e_v)) \times 1 + d_n(e_v) \times 2 = U_n + d_n(e_v),$$

In practice dependence distances are almost always 0 or 1, i.e., $d_n(e_v) < U_n$, allowing us to simplify the computation of register pressure using

$$R_r^C = \sum_{v \in V_r^C} \prod_{1 \le i < n} (U_i - d_i(e_v))^+ (U_n + d_n(e_v)).$$

(3) For each reference $v \in V_r^I$ that is invariant with respect to loop n, one register is required if loop n is unrolled. If v is not invariant with respect to n, unrolling n does not increase register pressure, but provides opportunities for scalar replacement that may be utilized if an outer loop j is unrolled and v is invariant with respect to loop j.

Carr and Kennedy have formularized the register pressure for unroll-and-jammed loops [21]. Combining their results and the above observation gives:

$$R_r^I = \sum_{v \in V_r^I} (\prod_{1 \le i \le n} \alpha(e_v, i, n)),$$

where

$\alpha(e,i,n) \Leftarrow$ if e is invariant wrt loop i then
    if $(\exists U_j | U_j > 1 \wedge e$ is invariant wrt loop $j)$ or
        (e is invariant wrt loop n) then
      return 1
    else
      return 0
  else
    if $(i \neq n)$ or
        $(\exists U_j | U_j > 1 \wedge e$ is invariant wrt loop $j)$
      return U_i
    else
      return 0 .

3.5 Computing Unroll Amounts

To find the best unroll amounts for a particular loop on a particular target architecture, the following integer-optimization problem is solved:

objective function: min $uMinII$

constraints: $R_r^C + R_r^I \leq R_m$
$U_a, U_p \geq 1$

where R_m is the number of physical registers in the target machine.[2]

To solve this problem, we can bound the search space and do an exhaustive search to find the best unroll amounts. Carr and Kennedy have shown that bounding the space by R_m in each dimension is sufficient [21]. If an exhaustive search is too expensive, the heuristic search algorithm shown in Figure 3.3 may be used. This algorithm attempts to find the optimal unroll amount based upon the type of a loop, which is defined below,

- A loop is called a *memory-bound* loop if ResII=$\lceil \frac{M}{FU_m} \rceil$.
- A loop is *communication-bound* if ResII=$\lceil \frac{C}{FU_c} \rceil$.

[2]Note that registers will need to be reserved to allow for the increase in register pressure caused by software pipelining.

- Otherwise the loop is a *compute-bound* loop.

Initially two loops l_a and l_p (they can be the same loop) are selected to unroll, as proposed in Section 3.2. The routine *ComputeUnrollAmounts* starts with $U_a = U_p = 1$, and uses a heuristic approach to reduce the unit ResII by adjusting the values of U_a and U_p according to the limiting factor that determines ResII. This step is repeatedly executed until the unit ResII cannot be improved or the register pressure exceeds the physical register size.

If a loop is compute-bound, unroll-and-jam or loop unrolling cannot remove the computation from the innermost loop body. In this case $ComputerUnrollAmounts$ simply generates copies of the loop body for multiple clusters by unrolling l_p loop with a factor less than or equal to the number of clusters as long as unrolling does not increase the unit ResII or yield too much register pressure.

The performance of a memory-bound loop can be improved by increasing the amount of data reuse via unroll-and-jam. Therefore *ComputeUnrollAmounts* tries a new II with $U_a + 1$ for memory-bound loops.

For a communication-bound loop, we consider the following cases:

1. $l_a = l_p$ The unit ResII is determined by:

$$uResII = \frac{\lceil \frac{C}{FU_c} \rceil}{U_a \times U_p}.$$

   The discussion in Section 3.3.1 shows that the number of intercluster copies for unrolling a single loop can be considered as a constant with respect to the unroll amount U_a. As a result a smaller unit ResII can be expected by increasing U_a.

2. $l_a \neq l_p$

   It is hard to predict the exact effect of changing U_a or U_p on communication costs. A heuristic is used for this case. First a smaller U_a is used to recompute a new ResII since unroll-and-jamming l_a fewer times can cause fewer dependences to become innermost, and hence is likely to reduce the intercluster communication. If a smaller U_a fails to decrease ResII, a smaller U_p is used to compute ResII with hope that fewer l_p-carried dependences will be brought into the innermost loop after unroll-and-jam or loop unrolling.

```
Procedure ComputeUnrollAmount()

U_a = U_p = 1
uResII = GetUnitResII(U_a, U_p, loopType)
oldResII = uResII
newResII = 0
while (oldResII > newResII and
      GetRegisterPressure(U_a, U_p)< R_m)
{
  if (loopType = compute_bound)
    if (U_p < # of clusters)
      U_p = U_p + 1
  else if (loopType = memory_bound)
    if (U_p < # of clusters)
    {
      U_p = U_p + 1
      newResII = GetUnitResII(U_a, U_p, loopType)
      if (newResII>oldResII or
          GetRegisterPressure(U_a, U_p)> R_m)
      {
        U_p = U_p − 1
        U_a = U_a + 1
      }
    }
    else
      U_a = U_a + 1
  else /* communication-bound loop*/
    if (l_a = l_p)
      U_a = U_a + 1;
    else
      if (U_a >2)
      {
        U_a = U_a − 1
        newResII = GetUnitResII(U_a, U_p, loopType)
        if (newResII>oldResII or GetRegisterPressure(U_a, U_p)> R_m)
          if (U_a >2)
            U_a= U_p-1
      }
      else if (U_p > 2)
        U_p= U_p-1
  newResII = GetUnitResII(U_a, U_p, loopType)
}
```

Figure 3.3: Compute Unroll Amounts

3.6 Summary

This chapter has presented a new method for performing unroll-and-jam and/or loop unrolling to achieve a high degree of ILP on a clustered VLIW machine. Instead of using fixed unroll amounts, this method automatically tailors unroll-and-jam/unrolling for a given loop nest based upon resources available on a specific architecture. This is done by solving an integer-optimization problem which predicts the performance of software pipelined loops and the register pressure after unroll-and-jam/loop unrolling and scalar replacement are applied. This method has been implemented in a research compiler and tested on DSP benchmarks. The results are reported in Chapter 5.

Chapter 4

Loop Fusion

This chapter demonstrates how loop fusion, used as a precursor to unrolling, can be used to improve loop performance on clustered VLIW architectures.

The profitability of fusion is computed by examining its effect on unrolling. By combining two (or more) loop nests into a single nest, operations from the second loop can be scheduled in any empty instruction slots that exist in the original loop. The resulting loop provides more parallelism than the two loops in isolation, reduces the effect of leakage power due to unused functional units, lessens the overhead due to loop code, and often takes less code space. However, loop fusion may degrade performance if the fused loop increases the amount of intercluster communication. This chapter describes a method that fuses as many loops as possible while retaining low communication overhead.

This chapter first introduces the safety of loop fusion, then presents a cost model for determining the profitability of fusion, followed by a slightly modified greedy fusion algorithm.

4.1 Safety of Loop Fusion

In this work, loop fusion is applied to compatible loops. Two loops are compatible if they have the same loop bounds and loop stride. (Note that the loops can have different loop indices.) If the trip counts of two loops differ by a small number t, peeling t iterations of the loop that has a larger trip count can generate a new loop with a compatible loop header.

If the source and sink of a dependence are located in the same loop, loop fusion

does not change their execution order. Therefore, only dependences between two loops are examined to determine the safety of loop fusion. If the loops to be fused have a common outer loop and the dependence is carried by the outer loop, reordering the execution order of the inner loops does not change the dependence carrier. The dependence constraints are satisfied through the execution of the outer loop, and hence it is safe to fuse the inner loops.

After fusion, loop-independent dependences will become (1) loop-independent dependences, (2) loop-carried forward dependences, or (3) loop-carried backward dependences. Cases (1) and (2) preserve the direction of dependences. On the other hand, loop fusion is illegal for the third case, because after fusion the execution order of the source and sink of a dependence is reversed, as shown in the following example. The original loops are:

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

and the fused loop will be

```
DO I = 1, N
  A(I) = B(I) + C(I)
  D(I) = A(I+1) + B(I)
ENNDO
```

The array `A` is defined in the first loop and is used in the second loop. In the fused loop, a loop-carried anti dependence is introduced and the value of an array element used on the current iteration will be assigned one iteration later. Therefore, loop fusion is not safe. A loop-independent dependence is fusion-preventing if the dependence becomes a backward loop carried dependence after fusion [1, 63].

When two loops are not adjacent, checking the dependences between two loops is not sufficient to determine the safety of fusion. The dependence paths connecting two loops must be considered. If a dependence path from the first loop to the second loop contains a statement or a loop that cannot be fused with either loop, two loops cannot be fused unless some transformations are performed to break the path (e.g., scalar renaming can be used to remove anti or output dependences [8, 50]). For instance, in the following loop, loop fusion is prevented because there is no way to preserve both loop-independent true dependences at the same time after fusion.

```
S = 0
DO I = 1, N
  S = S + A(I) * B(I)
ENDDO
T = X(S)
DO I = 1, N
  T = T + C(I)
ENDDO
```

In summary, two loops can be fused together legally if

- the loop headers are compatible,
- fusion does not convert a loop-independent dependence into a backward loop-carried dependence, and
- there is no dependence path between two loops that contains a statement or a loop that is being fused with them.

4.2 Communication Cost

Although loop fusion provides opportunities to make efficient use of clustered VLIW machines, careless use of loop fusion may hamper performance due to high intercluster communication. For example, given the following loops:

```
DO I = 1, N
   A(I) = X(I) + Q
ENDDO
DO I = 1, N
   B(I) = A(I) * C(I)
ENDDO
DO I = 1, N
   D(I) = A(I-1) + B(I)
ENDDO
```

fusing them together gives the loop

```
DO I = 1, N
   A(I) = X(I) + Q
   B(I) = A(I) * C(I)
   D(I) = A(I-1) + B(I)
ENDDO
```

This loop provides opportunities for data reuse involving references to `A(I)` and `B(I)`. However, extra alignment conflicts are introduced into the fused loop. One dependence path from the first statement to the third statement, due to references to `A(I)` and `B(I)`, has a total distance of 0, while a second dependence path between these two statements due to references to `A(I)` and `A(I-1)` has a distance of 1. This is an alignment conflict that may restrict intercluster parallelism when the loop is unrolled for multiple clusters.

To avoid introducing additional alignment conflicts, we can fuse the first two loops, or the second and third loop, instead of fusing all three together. The resulting loops can be unrolled to generate independent copies of statements that can be executed in parallel across multiple clusters.

The following example illustrates another case when loop fusion can adversely affect unroll-and-jam.

```
DO I = 1, N
  S = 0
  DO J = 1, M
    S = S + A(J) * B(J)
  ENDDO
  X(I) = S
ENDDO
DO I = 1, N
  C(I) = C(I-1) * Q
  DO J = 1, M
    D(J) = C(I) + A(J) * B(J)
  ENDDO
ENDDO
```

The `I`-loop in the first loop nest contains no alignment conflicts, indicating unroll-and-jam will be beneficial. This is not the case for the second loop nest where the `I`-loop carries a recurrence and makes unroll-and-jam unprofitable at this level. Instead, unrolling the `J`-loop in the second nest will produce independent copies of the loop body. In this example, fusing two loops together is not a good choice since either loop level of the fused loop will carry a recurrence that inhibits the ability of unroll-and-jam and unrolling to create data-independent intercluster parallelism.

Based upon the above discussion, loop fusion is applied when the fused loop does not increase the number of alignment conflicts. To compute this, a notion of communication cost is first introduced. Communication cost at level l of a loop nest L, $C_L(l)$, is defined

```
Procedure FusionIsProfitable(L1, L2)
Input: L1, L2: original loops

compute C_L1, C_L2
compute Min_L1, Min_L2
compute C_Lf for the fused loop Lf
commonLevels = Min_L1 ∩ Min_L2
if commonLevels ≠ φ
  if ∃ l ∈ commonLevels and
    C_Lf(l) = C_L1(l) + C_L2(l)
    return true;
  return false;
```

Figure 4.1: Determine Profitability of Loop Fusion

as the number of alignment conflicts at that level. For a loop nest L, the set of loop levels containing the least communication cost among communication costs at all levels is called the set of minimal loop levels for L, denoted Min_L.

Figure 4.1 shows the algorithm for determining whether loop fusion is profitable. The algorithm first computes the communication cost for original loops and the fused loop, and finds common loop levels at which both loops have minimal communication costs. These loop levels are good candidates for applying unroll-and-jam or unrolling during future transformations. The algorithm returns true if the communication cost of the fused loop at one of these levels is equal to the sum of the communication costs of the original loops at the same level, i.e., no extra alignment conflicts are introduced into the fused loop at this level.

It is worth noting that legal fusion does not introduce new recurrences into the fused loop, making this transformation more attractive.

4.3 Implementing Fusion

It has been shown that finding the optimal solution for a global fusion problem is NP-hard, and in practice a greedy heuristic for applying fusion is simple and sufficient

for finding good solutions [38]. This work performs loop fusion based upon the greedy algorithm proposed by Kennedy and McKinley in [38] using the algorithm in Figure 4.1 to evaluate profitability.

The approach proposed by Kennedy and McKinley begins by constructing a fusion graph, where each node n represents a loop, and an edge (n_1, n_2) is added between two nodes if there exists a dependence between two corresponding loops. Then the legality of loop fusion is examined. An edge (n_1, n_2) is marked fusion-preventing if n_1 and n_2 cannot be fused safely. Finally, the approach iteratively selects two loops and fuses them together if fusion is legal and beneficial until no such loop pair can be found.

Originally the communication cost is computed for each node and each edge as if the sink and source node of the edge were in the same loop. After fusion, two fusible nodes are collapsed into a single node, and the edges that connect the two old nodes are also collapsed. The communication cost of the new node is equal to the sum of communication costs of the two old nodes. Unlike the fusion problem for improving data locality where weights for collapsed edges can be calculated from the original graph, in this fusion problem, the communication costs for the edges into and out of the fused node need to be re-computed based upon the dependences between the fused loop and associated loops. This is because the original graph only records communication costs for fusing loop pairs, but has no information for fusing more than two loops into one. Fusion may introduce new alignment conflicts that were not detected when the original graph was constructed. Consider for instance the first example in the last section, the communication costs for three edges connecting original loops are all equal to zero. Fusing the first two loops together gives a communication cost of one to the collapsed edge. This value is computed base upon dependences between the fused loop and the third loop.

4.4 Summary

This chapter has demonstrated that loop fusion can provide more parallelism and improve the utilization of functional units, and therefore, is an important transformation for generating code with high quality on clustered VLIW architectures. A new metric has been developed to determine the profitability of loop fusion. The fusion algorithm based upon this metric has been implemented in a research compiler. The next chapter presents the results of applying loop fusion to DSP benchmarks on different clustered VLIW architectures.

Chapter 5

Experimental Results

The preceding chapters have developed new loop transformation methods to achieve a high degree of parallelism and reduce communication overhead on clustered architectures. This chapter evaluates those techniques using DSP benchmarks and five different architectures. Section 5.1 introduces the benchmarks and architectures used in this experiment. Section 5.2 reports the results achieved by the unroll-and-jam algorithm. Section 5.3 presents the improvement obtained via loop fusion.

5.1 Benchmarks and Configurations

The proposed algorithms have been implemented in Memoria [16], a source-to-source FORTRAN transformer based upon the DSystem [2]. The benchmark suite for this experiment consists of 119 loops that have been extracted from a DSP benchmark suite provided by Texas Instruments. The suite, including two full applications and 48 DSP kernels, contains typical DSP applications: FIR filter, correlation, Reed-Solomon decoding, lattice filter, LMS filter, etc. The DSP benchmarks were converted from C into FORTRAN by hand so that Memoria could process them. After conversion, each C data type or operation is replaced with a similar FORTRAN data type or operation. For example, unsigned integers are converted into integers in the FORTRAN code and bitwise operations in C are converted into the corresponding bitwise intrinsics in FORTRAN. By defining the operation cycle counts properly, this conversion allows accurate results to be achieved from this experiment.

To evaluate the effectiveness of the transformations, a simulated architecture,

called the URM [51], and the Texas Instruments TMS320C64x were used. For the URM, the code generated by Memoria has been translated into intermediate code and then compiled with Rocket [59], a retargetable compiler for ILP architectures. Rocket performs cluster partitioning [32], software pipelining [52], and Chaitin/Briggs style register assignment [12].

In this experiment, Rocket targets four different clustered VLIW architectures with the following configurations:

1. 8 functional units with 2 clusters of size 4
2. 8 functional units with 4 clusters of size 2
3. 16 functional units with 2 clusters of size 8
4. 16 functional units with 4 clusters of size 4

Each cluster has 48 integer registers. All functional units are homogeneous and have instruction timings as shown in Table 5.1. Intercluster copy operations do not use the instruction slots in the functional units. Instead, extra issue slot(s) are reserved for copies. The 8-wide machines support one copy unit and the 16-wide machines support two. As a result, each machine with 8 functional units can perform one copy between register fines in a single cycle, while the 16 functional unit machines can perform two per cycle.

Operations	**Cycles**
integer copies	1
float copies	1
loads	5
stores	1
integer mult.	2
integer divide	12
other integer	1
other float	2

Table 5.1: Operation Cycle Counts

In addition to the results for the URM, the proposed algorithms have been evaluated on the Texas Instruments TMS320C64x (or C64x). The C64x CPU is a two-cluster VLIW fixed-point processor with eight functional units that are divided equally between the clusters. Each cluster is directly connected to one register file having 32 general purpose registers. All eight of the functional units can access the register file in their own cluster

directly, and the register file in another cluster through a cross path. Since only two cross paths exist, a total of up to two cross path source reads can be executed per cycle. Other intercluster value transfers have to be done via explicit copy operations. On the C64x, multiplication instructions have one delay slot, load instructions have four delay slots, and branch instructions have five delay slots. Most other instructions have zero delay slots, while some can have up to three delay slots [60].

To make use of the C64x C compiler, the transformed code generated by the Memoria were converted into C by hand. Then, the C64x compiler was applied to both the original and transformed versions of the code, using the highest optimization level (-o3) [61].

Section 5.2 reports the results achieved by the unroll-and-jam algorithm on the URM and the TI C64x. Section 5.3 presents the improvement obtained via loop fusion. For each architecture, the speedups are evaluated by comparing the code generated by the new algorithms with the untransformed code. The previous work has shown that using the partitioning method presented in [32], one can expect 10-20% degradation in execution time when compared to an ideal monolithic-register architecture with the same level of ILP.

5.2 Unroll-and-jam Results

Out of the 119 loops in the benchmark suite, unroll-and-jam or loop unrolling is applicable to 71 loops. The results are reported over these 71 loops. Other loops contain function calls or conditional statements. Since Rocket cannot software pipeline such loops, loop unrolling was not performed on them. Section 5.2.1 reports the speedups achieved by Memoria on four clustered VLIW architectures modeled with the URM. Section 5.2.2 presents the speedups achieved by Memoria on the Texas Instruments TMS320C64x.

5.2.1 URM Results

The URM results for individual loops are presented in Tables A.1, A.2, A.3, and A.4, and are summarized in Tables 5.2 and 5.3. The results reflect the effect of the new unroll-and-jam method using heuristic search over 71 loops. Table 5.2 shows the average speedup obtained by unroll-and-jam (and/or loop unrolling) only. Table 5.3 reports the combined improvement of loop unrolling/unroll-and-jam) and scalar replacement over the original untransformed loops. The speedup is measured using the actual unit II (II per un-

rolled iteration) of the software pipelined loop before and after loop transformations. The rows labeled "Average" show the average speedup in unit II obtained by the transformed loops after unrolling (either inner- or outer-loop unrolling) and scalar replacement are applied, when compared with the unit II of loops without the transformations. The rows labeled "Harmonic," "Median," and "Std Dev" give the harmonic mean speedup in unit II, the median speedup in unit II and the standard deviation, respectively. The "Improved" row shows the number of loops that gain improvement via unroll-and-jam.

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
	Speedup			
Average	1.61	2.02	1.55	1.79
Harmonic	1.33	1.60	1.34	1.35
Median	1.52	1.60	1.53	1.60
Std Dev	0.93	1.35	0.69	1.17
Improved	50	69	50	51

Table 5.2: URM Speedups: Unrolled vs. Original

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
	Speedup			
Average	1.91	2.51	1.87	2.38
Harmonic	1.39	1.68	1.40	1.43
Median	1.52	1.78	1.60	1.60
Std Dev	1.72	2.52	1.65	2.58
Improved	50	69	50	51

Table 5.3: URM Speedups: Transformed vs. Original

The results show that the new unroll-and-jam algorithm improves the performance of 71 loops by an average harmonic mean of 1.33 – 1.60. The speedups for individual loops range from 0.69 to 8.00, with more than 50 loops, or 42% of loops in the benchmark suite, seeing improvement by the unroll-and-jam algorithm. Loop unrolling/unroll-and-jam and scalar replacement together achieved a 1.39 – 1.68 harmonic mean speedup in unit II over the original loops. The median speedup ranges from 1.52 – 1.78. The speedups for individual loops range from 0.7 to 14.4.

As can be seen in the range of speedups, some loops have very large speedups

and some loops actually show a degradation. Loop 26, 29 and 47 gain large speedups via unroll-and-jam. These loops are doubly nested and compute a reduction. Unroll-and-jam improves the performance of this type of loop particularly well. Unroll-and-jam also provide opportunities for scalar replacement for some loops, e.g., loop 16, 26, and 32. These loops contain outer-loop carried reuse that can be moved to the innermost loop level by unroll-and-jam and become amenable to scalar replacement. The transformed code for these loops appear excellent intracluster and intercluster parallelism.

Loop 22, 24 and 25 do not achieve performance improvement on any architectures. The index arrays in these loops make accurate dependence analysis nearly impossible. Because the dependence analysis is inexact, the prediction scheme does not correctly predict the communication needed by the loop. The performance degradation shown for other loops are due to the increase in RecII after unrolling. This can be attributed to the heuristic feature of Rocket.

Although the architecture with 8 functional units arranged in 4 clusters achieves the best overall speedup, the performance of each individual loop is lower than on the other architectures. In addition, the value of standard deviation suggests that speedups of individual loops on 4-cluster architecture vary more widely than on 2-cluster architectures. This is because some loops that have good parallelism can be unrolled more times to generate higher ILP on architectures with four clusters; other loops, however, gain small speedups due to a large number of intercluster data copies since having four clusters increases communication.

Using a fixed unroll amount, as is done in [34], may cause performance degradation when communication costs are dominant. Table A.6 shows the performance difference between the new unroll-and-jam algorithm and Huang's method [34] when the methods use different unroll amounts. The results are summarized in Table 5.4. The row marked "# of Loops" shows how many of loops have different unroll amounts under both methods on each clustered architecture. The row labeled "Harmonic Mean" gives the harmonic mean speedup in unit II obtained by the new method. The row labeled "Harmonic (Fixed)" shows the harmonic mean speedup in unit II obtained by Huang's method. The new algorithm gives a better harmonic mean speedup in unit II than Huang's method on each architecture. The degradations seen on the 4-cluster 8-wide machine are due to the dependences with indeterminate dependence distances caused by index arrays.

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
# of Loops	9	4	9	21
	Speedup			
Harmonic Mean	1.00	0.91	1.00	1.07
Harmonic (Fixed)	0.88	0.84	0.88	0.95

Table 5.4: URM Speedups: The New Algorithm vs. Fixed Unroll Amounts

5.2.2 TMS320C64x Results

The transformed versions of the code were run on the C64x and the resulting IIs were compared with the original code. Since the unrolling scheme in the C64x compiler is a subset of Huang's algorithm, unrolling in the TI complier was turned off to examine the effect of the new unrolling algorithm on the set of loops. The new algorithm was compared to Huang's later in this section.

Table 5.5 summarizes the results obtained on the C64x, and the results of individual loops are listed in Table A.5. The C64x compiler failed to find a profitable schedule for two loops, and thus generated code for these loops without applying software pipelining. Additionally two loops that contain a division operation cannot be software pipelined either since the C64x compiler treats division operations as function calls that disable software pipelining. Therefore, Table 5.5 and Table A.5 gives the results for 67 loops of the benchmark suite.

	Speedup
Average	1.94
Harmonic	1.70
Median	2.00
Std Dev	0.74
Improved	57

Table 5.5: TMS320C64X Speedups: Unrolled vs. Original

On the C64x, all cases showed improvement or remained unchanged with unrolling. The harmonic mean speedup in unit II across the entire benchmark suite is 1.7. The individual speedups for the loops ranged from 1.0 to 4. The harmonic mean speedup improvement is better than that seen on the URM. The C64x supports more intercluster copies which, in turn, reduces the overhead of the copy operations.

On the 9 loops where the new algorithm produces a different result than Huang's algorithm, Huang's algorithm gets better performance. This is because the C64x supports SIMD operations and the new performance model does not considers these effects. The new model will choose not to unroll these loops because of communication. However, unrolling allows the C64x compiler to detect the SIMD operations and an improvement is obtained. Since Huang's algorithm always unrolls irrespective of communication cost, it blindly exposes the SIMD operations. The solution to this problem is to model SIMD operations in the performance model.

5.2.3 Accuracy of Communication Cost Prediction

To evaluate the accuracy of the unroll-and-jam algorithm in predicting intercluster communication, we compare the number of intercluster data transfers due to cross-cluster dependences predicted by the communication cost model against the actual number of cross-cluster dependences found in the transformed loops. For each loop after unroll-and-jam/unrolling and scalar replacement are applied, Memoria counts the intercluster true dependences whose sinks are not killed by a definition in the path from the source to the sink. The input dependences whose sources and sinks reside in distinct clusters are also recorded. Table A.7 lists the number of predicted copies and the number of intercluster dependences for each individual loops on four clustered architectures.

The results show that the communication cost model makes an exact prediction for most of the loops in the benchmark suite. For the 2-cluster machines, only 5 out of 71 loops show a misprediction. For the 4-cluster machines, 7 loops have a misprediction. In each case expect for loop 22 and 25, the misprediction is by one, two, or three dependences and occurs in the loops that contain inconsistent dependences. Loop 22 and 25 show a large misprediction due to the index arrays in these loops.

Many heuristic algorithms may not be able to derive a partition that limits copies to the extent that the cost model predicts. However, this prediction serves as a lower bound for these heuristics to attempt to achieve.

5.3 Fusion results

Out of the 119 loops in the benchmark suite, fusion is applicable to 12 sets of loops consisting of 25 total loops. Table 5.6 describes the main functions of those loops.

Loop Set #	**Description**
1,2	solves the error locater polynomial equation
3	scales up or down a line of data
4	performs 2D wavelet transform on input data
5,6,7,8,9	performs shift and dot product on vectors
10	performs multiplication, accumulation, and subtraction on an array and finds the largest value
11	performs dot product on vectors
12	transposes the data to get pixel planes

Table 5.6: Loop Description

Section 5.3.1 reports the speedups achieved by Memoria on four clustered VLIW architectures modeled with the URM. Section 5.3.2 presents the speedups achieved by Memoria on the Texas Instruments TMS320C64x.

5.3.1 URM Results

Tables 5.7, 5.8 and 5.9, show the effects of loop fusion and loop unrolling (either inner- or outer-loop unrolling) on the 12 sets of loops. (For the results on individual loops, see Tables B.1, B.2, B.3, and B.4.) The rows labeled "Average" show the average speedup in unit II obtained by the transformed loops when compared with the unit II of loops without the transformations. For the unfused loops the sum of the original unit IIs was compared to the unit II of the fused loop. The rows labeled "Harmonic," "Median," and "Std Dev" give the harmonic mean speedup in unit II, the median speedup in unit II, and the standard deviation, respectively.

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
	Speedup			
Average	1.59	1.65	1.66	1.59
Harmonic	1.48	1.56	1.56	1.48
Median	1.38	1.51	1.90	1.38
Std Dev	0.45	0.45	0.40	0.45

Table 5.7: URM Speedups: Fused vs. Original

Table 5.7 shows the effects of performing only loop fusion on the benchmark suites. Loop fusion has gained a 1.48 – 1.56 harmonic mean speedup in unit II over the four target architectures. The speedups on individual loops range from 1.14 to 2.67. For most of these tests, the architecture configuration made little difference. This is because the original loops do not have enough parallelism to occupy all of the issue slots.

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
	Speedup			
Average	1.57	1.58	1.55	1.48
Harmonic	1.49	1.41	1.47	1.29
Median	1.54	1.65	1.58	1.27
Std Dev	0.41	0.54	0.37	0.54

Table 5.8: URM Speedups: Fused & Unrolled vs. Unrolled Only

Table 5.8 shows the improvements obtained by loop fusion over unrolling. Again, we see significant gains with a 1.29 – 1.49 harmonic mean speedup in unit II. The speedups on individual loops ranged from 0.61 to 2.67, with the median speedup ranging from 1.27 to 1.65. A degradation in unit II was observed for one test case (loop set 11) on 4-cluster architectures. This is because the fused loop contains significant amount of computation which causes high communication overhead on architectures with more clusters. On the other hand, the original loops are relatively small and can create more parallelism via loop unrolling.

Finally, Table 5.9 reports the total improvement of fusion and unrolling over the original untransformed loop. We have observed a 1.72 – 1.89 harmonic mean speedup in unit II and a median speedup of 1.94 – 2. The speedups for individual loops ranged from 1 to 3.05.

Width	**8 FUs**		**16 FUs**	
Clusters	2	4	2	4
	Speedup			
Average	1.92	2.09	1.94	1.89
Harmonic	1.82	1.89	1.86	1.72
Median	1.94	2	2	1.9
Std Dev	0.43	0.68	0.35	0.57

Table 5.9: URM Speedups: Fused & Unrolled vs. Original

The largest speedup was seen in the architecture with 8 functional units arranged in 4 clusters. However, the performance of individual loops is lower than on other architectures. This is because this architecture has enough functional units to capture parallelism exploited by unrolling, but requires a significant number of intercluster data transfers since only one such transfer can be initiated in a single cycle. When the performance of individual loops and the average overall speedups are considered, the architectures with fewer partitions can perform better. The reason is that such architectures offer a large amount of ILP with a relatively low communication overhead since there are only two clusters.

The results for the URM show that loop fusion offers tremendous potential for improving code on clustered VLIW architectures. Fusion is applicable to 21% of the loops found in the benchmark suite and, when it is applied, significant improvements are obtained.

5.3.2 TMS320C64x Results

The TI compiler already performs unrolling, so our measurements only show the effects of loop fusion. Table 5.10 summarizes the results obtained on the C64x, while Table B.5 reports the results for individual loops.

	Speedup
Average	1.46
Harmonic	1.33
Median	1.33
Std Dev	0.46

Table 5.10: TMS320C64x Speedups: Fused vs. Original

On the C64x, fusion achieves an average speedup in unit II of 1.46, a harmonic mean speedup of 1.33, and a median speedup of 1.33. The speedups ranged from 1 to 2 with four of the loop sets giving a speedup of 2. The C64x unrolls four loop sets entirely and hence no improvement was gained with fusion for these four loop sets. The mean speedup improvement on all loop sets is less than that seen on the URM. The difference in speedup and the lack of any speedup in four of the loops can be attributed to the availability of cross paths on C64x. The cross paths allow communication between clusters without an explicit copy operation. This reduces the overhead of the communication compared to the URM and offers fewer opportunities for improvements.

5.4 Summary

This chapter has tested the proposed loop transformation strategy on different kinds of clustered architectures. The results show that

- Unroll-and-jam and loop unrolling are very effective for the DSP benchmark suite used in this research, improving more than 40% of loops with an average harmonic mean speedup of 1.33 – 1.60.

- Unroll-and-jam and scalar replacement can generate high quality of code with enhanced intracluster and intercluster parallelism, giving an average harmonic mean speedup of 1.39 – 1.68.

- Loop fusion is applicable to 21% of the loops found in the benchmark suite and, when it is applied, a 1.33 – 1.56 harmonic mean speedup is obtained.

- Combined optimizations of loop fusion, loop unrolling, and scalar replacement offer tremendous potential for improving code on clustered VLIW architectures. A speedup of 1.72 – 1.89 can be expected through the combined optimization strategy.

Chapter 6

Conclusions and Future Work

This dissertation has studied the effects of high-level loop transformations on clustered VLIW architectures. The goal is to improve code generation to make efficient use of clustered architectures. This chapter summarizes the new techniques developed in the preceding chapters and then discusses the future work of this research.

6.1 Contribution

Loop Fusion This dissertation is the first work that investigates the effect of loop fusion on clustered VLIW architectures. Loop fusion enhances instruction-level parallelism, generates compact code and improves the utilization of functional units. The method developed in Chapter 4 employed alignment conflicts as a metric for guiding loop fusion and preventing aggressive fusion that would restrict intercluster parallelism created by unroll-and-jam. A greedy loop fusion algorithm has been implemented and tested on DSP benchmarks for four different simulated, clustered VLIW architectures, and the TI C6x. The results show that, out of 119 DSP benchmarks, loop fusion is applicable to 21% of the loops, and can result in a 1.33 – 1.56 harmonic mean speedup in the unit II. Fusion and unrolling together gain a 1.72 – 1.89 harmonic mean speedup on four simulated architectures.

Unroll-and-jam and Loop Unrolling Unroll-and-jamming a loop nest can improve the utilization of functional units in a single cluster. This research also uses unroll-and-jam and loop unrolling to generate intercluster parallelism while maintaining low communication overhead. Chapter 3 presents a new method for predicting the amount of intercluster

data transfers caused by data dependences in loops run on clustered VLIW architectures. This method is part of an integer-optimization problem used to predict the performance of software-pipelined loops and to guide unroll-and-jam or loop unrolling for clustered VLIW architectures.

The results show that the prediction of intercluster data dependences that cause communication is accurate. In addition, out of the 119 DSP benchmarks used, 42%-58% of the loops can be improved via unroll-and-jam/unrolling and scalar replacement by a harmonic mean speedup of 1.39 – 1.68 for four different simulated, clustered VLIW architectures, and a harmonic mean speedup of 1.7 for the TI TM320C64x. In contrast to previous work that simply uses fixed unroll amounts, this method tailors unroll-and-jam/loop unrolling for specific loops based upon the parameters of specific architectures, and therefore, can generate more efficient code.

6.2 Future Work

This research confirms the abilities of high-level loop transformations to improve performance for clustered VLIW architectures, and it paves the way for much more compiler research.

Define a Unified Metric for Loop Transformations This research determines the profitability of loop fusion based upon the impact of fused loops on parallelism caused by unroll-and-jam. The algorithm that computes the communication cost can be extended to examine whether the extra data transfers can be hidden in software pipelined loops. This can result in a unified metric based upon the performance prediction for guiding loop fusion and unroll-and-jam.

Overcome Inaccurate Dependence Analysis The effectiveness of the unroll-and-jam algorithm highly depends on the accuracy of the prediction of intercluter communication. Although dependence analysis is enough for accurate prediction for most cases, dependence patterns such as index arrays and variant dependence distances do exist in DSP benchmarks, making exact dependence analysis very difficult. Integrating the prediction of intercluster copies with an improved dependence testing method that deals with complex dependence patterns would generate better unroll-and-jammed loops.

Study the Effects of Scalar Replacement The loop transformation strategy described in Chapter 2 uses scalar replacement to enhance the effectiveness of loop fusion and unroll-and-jam. The results show that some loops obtain large improvement via unrolling and scalar replacement, while others gain small improvement or even degradations in performance. An open question is: how will scalar replacement affect clustered VLIW architectures? Scalar replacement eliminates redundant memory accesses by keeping array elements in registers. However, it increases register pressure and possibly communication overhead, making it difficult to produce a high degree of intercluster parallelism. A compiler must make tradeoffs between effective register use and a high level of intercluster parallelism when performing scalar replacement.

Investigate Pre-partitioning This dissertation has verified the abilities of high-level transformations to expose more ILP available to the low-level optimizer. However the low-level optimizer may not be able to make full use of parallelism due to lack of information on parallelism in the intermediate code. A remedy is to perform pre-partitioning at a high-level after loop transformations are applied. High-level pre-partitioning distributes the statements of the transformed loop among clusters according to predicted intercluster communication overhead. The resulting partitions can be used to improve register partitioning at the low-level. For instance, the low-level optimizer can use the high-level partitions as the initial partitioning scheme, or as a tie-breaker when multiple clusters have equal priority to hold a value.

Investigate Other Objectives The goal of loop transformations in this research is to improve performance of applications run on clustered VLIW architectures. This research can be extended to achieve other objectives. For instance, some clustered VLIW architectures provide SIMD, or the single-instruction multiple-data techniques that execute multiple instances of the same operations in parallel using different data. Unroll-and-jam or loop unrolling can be applied to provide opportunities of creating SIMD instructions. In addition, new metrics can be adopted for guiding loop transformations, such as, making efficient use of functional units or reducing the code size after transformations. Such an extension would be extremely important for those embedded system applications that are sensitive to power consumption and memory use as well as performance.

6.3 Final Remarks

Using compiler techniques to take advantage of target architectures has a long history. Clustered VLIW embedded processors are increasing in use, making it important for compilers to achieve high ILP with low communication overhead. The work presented in this dissertation provides automatic techniques that free programmers from optimizing programs manually. This research is an important step in generating high performance code for clustered architectures, indicating that high-level loop transformations should be an integral part of compilation for clustered VLIW machines.

Appendix A

Individual Loop Performance for Unrolling

Table A.1, A.2, A.3, and A.4 present the URM results for individual loops when loop unrolling (and/or unroll-and-jam) and scalar replacement are applied. In these tables,

Loop# is the loop number,

Org II is the II for the original loop,

Unroll uII is the unit II for the unrolled loop,

UnrollSR uII is the unit II after loop unrolling/unroll-and-jam and scalar replacement are applied,

Unroll vs Org is the speedup of the unrolled unit II compared with the orginal II,

UnrollSR vs Org is the speedup of the unit II for the transformed loop compared with the original II,

Avg is the average mean speedup in unit II,

Har is the harmonic mean speedup in unit II,

Med is the median speedup in unit II,

SD is the standard deviation.

Loop#	**Org II**	**Unroll uII**	**UnrollSR uII**	**Speedup**	
				Unroll vs Org	**UnrollSR vs Org**
1	9	4.5	4.5	2	2
2	14	9.5	5	1.47	2.8
3	15	8	8	1.875	1.875
4	2	1	1	2	2
5	2	1.167	1.167	1.71	1.71
6	2	1.167	1.167	1.71	1.71
7	2	1.167	1.167	1.71	1.71
8	5	4	4	1.25	1.25
9	2	2.75	2.5	0.73	0.8
10	7	5.167	5.167	1.35	1.35
11	2	2.75	2.5	0.73	0.8
12	2	2.75	2.5	0.73	0.8
13	2	1	1	2	2
14	2	2	2	1	1
15	2	1	1	2	2
16	21	9.5	2	2.21	10.5
17	4	3	3	1.33	1.33
18	2	1	1	2	2
19	3	1.5	1.5	2	2
20	15	8	7	1.875	2.14
21	11	5.5	5.5	2	2
22	28	28	28	1	1
23	34	17	17	2	2
24	8	8	8	1	1
25	32	33.5	33.5	0.96	0.96
26	21	5	2.5	4.2	8.4
27	23	21	10.5	1.10	2.19
28	13	8.5	6.5	1.53	2
29	3	0.5	0.5	6	6
30	5	2.833	2.333	1.769	2.14
31	5	2.5	2.5	2	2
32	18	8.5	2.5	2.12	7.2
33	5	2.5	2.5	2	2
34	8	5	5	1.6	1.6
35	2	1	1.5	2	1.33
36	3	2.5	2.5	1.2	1.2

Loop#	Org II	Unroll uII	UnrollSR uII	Speedup	
				Unroll vs Org	UnrollSR vs Org
37	2	1.167	1.167	1.71	1.71
38	2	1.167	1.167	1.71	1.71
39	15	8	8	1.875	1.88
40	2	2.5	2.5	0.8	0.8
41	5	4	4	1.25	1.25
42	5	4	4	1.25	1.25
43	2	2	2	1	1
44	2	2.5	2.5	0.8	0.8
45	3	2.333	2.333	1.29	1.29
46	3	2.5	2.5	1.2	1.2
47	3	0.5	0.5	6	6
48	2	2	2	1	1
49	2	2.5	2.5	0.8	0.8
50	5	3	3	1.67	1.67
51	2	2	2	1	1
52	2	2.5	2.5	0.8	0.8
53	2	2.5	2.125	0.8	0.94
54	3	2	2	1.5	1.5
55	5	4	4	1.25	1.25
56	5	4	4	1.25	1.25
57	2	1	1	2	2
58	2	2	2	1	1
59	9	5	5	1.8	1.8
60	5	3	3	1.67	1.67
61	5	4	4	1.25	1.25
62	2	2	2	1	1
63	5	3	3	1.67	1.67
64	2	2	2	1	1
65	5	3	3	1.67	1.67
66	2	2	2	1	1
67	8	5.25	5.25	1.52	1.52
68	16	10	5	1.6	3.2
69	3	3	3	1	1
70	3	1.5	3	2	1
71	2	2.5	2.5	0.8	0.8
Avg				1.61	1.91
Har				1.33	1.39
Med				1.52	1.52
SD				0.93	1.72

Table A.1: Unrolling Results for Individual Loops on 8-wide 2-cluster

Loop#	**Org II**	**Unroll uII**	**UnrollSR uII**	**Speedup**	
				Unroll vs Org	**UnrollSR vs Org**
1	9	4.25	4.25	2.12	2.12
2	14	10.5	1.25	1.33	11.2
3	15	7.25	7.25	2.07	2.07
4	3	0.75	0.75	4	4
5	3	1.875	1.75	1.6	1.71
6	3	1.875	1.75	1.6	1.71
7	3	1.875	1.5	1.6	2
8	5	4.25	4.25	1.18	1.18
9	3	2.667	2.5	1.12	1.2
10	7	5.583	5.583	1.25	1.25
11	3	2.667	2.5	1.12	1.2
12	3	2.667	2.5	1.12	1.2
13	3	0.75	0.75	4	4
14	4	1.75	1.75	2.29	2.29
15	3	0.75	0.75	4	4
16	21	4.75	2	4.42	10.5
17	4	2.25	2.25	1.78	1.78
18	3	0.75	0.75	4	4
19	3	1	1	3	3
20	15	8	7.5	1.88	2
21	11	5.25	5.25	2.09	2.09
22	29	26.5	26.5	1.09	1.09
23	34	13.75	13.75	2.47	2.47
24	8	11.667	11.66	0.69	0.69
25	32	33.5	33.5	0.96	0.95
26	21	5	2.5	4.2	8.4
27	23	21.75	11.75	1.06	1.96
28	13	9.25	7.25	1.40	1.79
29	3	0.375	0.375	8	8
30	5	2.75	2.75	1.82	1.82
31	5	2.5	2.5	2	2
32	18	4.5	1.25	4	14.4
33	5	2.5	2.5	2	2
34	8	4.5	4.5	1.78	1.78
35	3	1.625	1.625	1.85	1.85
36	5	3.75	3.75	1.33	1.33

Loop#	Org II	Unroll uII	UnrollSR uII	Speedup	
				Unroll vs Org	UnrollSR vs Org
37	3	2	2	1.5	1.5
38	3	2	2	1.5	1.5
39	15	6	6	2.5	2.5
40	3	2.417	2.417	1.24	1.24
41	5	4.25	4.25	1.18	1.18
42	5	4.25	4.25	1.18	1.18
43	4	2.25	2.25	1.78	1.78
44	3	2.5	2.5	1.2	1.2
45	4	2.5	2.5	1.6	1.6
46	4	2.25	2.25	1.78	1.78
47	5	0.625	0.625	8	8
48	3	3	3	1	1
49	3	2.5	2.5	1.2	1.2
50	5	2.5	2.5	2	2
51	3	2.5	2.5	1.2	1.2
52	3	2.5	2.5	1.2	1.2
53	3	2.5	2.5	1.2	1.2
54	5	2.25	2.25	2.22	2.22
55	5	4.25	4.25	1.18	1.18
56	5	4.25	4.25	1.18	1.18
57	3	1	1.125	3	2.67
58	3	2.5	2.5	1.2	1.2
59	9	3.5	3.5	2.57	2.57
60	5	2.5	2.5	2	2
61	5	4.25	4.25	1.18	1.18
62	3	2.5	2.5	1.2	1.2
63	5	2.5	2.5	2	2
64	3	2.5	2.5	1.2	1.2
65	5	2.5	2.5	2	2
66	3	2.5	2.5	1.2	1.2
67	8	5.167	5.167	1.54	1.54
68	16	9.75	4.25	1.64	3.76
69	3	1.25	1.25	2.4	2.4
70	3	1.75	2.5	1.71	1.2
71	3	2.5	2.5	1.2	1.2
Avg				2.02	2.51
Har				1.60	1.68
Med				1.6	1.78
SD				1.35	2.52

Table A.2: Unrolling Results for Individual Loops on 8-wide 4-cluster

Loop#	**Org II**	**Unroll uII**	**UnrollSR uII**	**Speedup**	
				Unroll vs Org	**UnrollSR vs Org**
1	9	4.5	4.5	2	2
2	14	9.5	5	1.47	2.8
3	15	8	8	1.86	1.86
4	2	1	1	2	2
5	2	1.25	1.25	1.6	1.6
6	2	1.25	1.25	1.6	1.6
7	2	1.25	1.25	1.6	1.6
8	5	4	4	1.25	1.25
9	2	2.833	2.333	0.71	0.86
10	7	5.75	5.75	1.22	1.22
11	2	2.833	2.333	0.71	0.86
12	2	2.833	2.333	0.71	0.86
13	2	1	1	2	2
14	2	1	1	2	2
15	2	1	1	2	2
16	21	9.5	2	2.21	10.5
17	4	3	3	1.33	1.33
18	2	1	1	2	2
19	3	1.5	1.5	2	2
20	15	8	7	1.88	2.14
21	11	5.5	5.5	2	2
22	28	28	28	1	1
23	34	17	17	2	2
24	8	8	8	1	1
25	32	33.5	33.5	0.96	0.96
26	21	4.833	2.333	4.34	9.00
27	23	21.5	10.5	1.07	2.19
28	13	8.5	6.5	1.53	2
29	3	0.75	0.75	4	4
30	5	3	2.25	1.67	2.22
31	5	2.5	2.5	2	2
32	18	8.5	2.5	2.12	7.2
33	5	2.5	2.5	2	2
34	8	5	5	1.6	1.6
35	2	1	1	2	2
36	2	2	2	1	1

Loop#	Org II	Unroll uII	UnrollSR uII	Speedup	
				Unroll vs Org	UnrollSR vs Org
37	2	1.25	1.25	1.6	1.6
38	2	1.25	1.25	1.6	1.6
39	15	8	8	1.88	1.87
40	2	2.333	2.333	0.86	0.85
41	5	4	4	1.25	1.25
42	5	4	4	1.25	1.25
43	2	2	2	1	1
44	2	2.333	2.333	0.86	0.85
45	3	2.25	2.25	1.33	1.33
46	3	2.5	2.5	1.2	1.2
47	3	0.75	0.75	4	4
48	2	2	2	1	1
49	2	2.333	2.333	0.86	0.86
50	5	3	3	1.67	1.67
51	2	2	2	1	1
52	2	2.333	2.333	0.86	0.86
53	2	2.333	2.333	0.86	0.86
54	2	1.5	1.5	1.33	1.33
55	5	4	4	1.25	1.25
56	5	4	4	1.25	1.25
57	2	1	1	2	2
58	2	2	2	1	1
59	9	5	5	1.8	1.8
60	5	3	3	1.67	1.67
61	5	4	4	1.25	1.25
62	2	2	2	1	1
63	5	3	3	1.67	1.67
64	2	2	2	1	1
65	5	3	3	1.67	1.67
66	2	2	2	1	1
67	8	5.333	5.333	1.50	1.50
68	16	10	5	1.6	3.2
69	3	3	3	1	1
70	3	1.5	3	2	1
71	2	2.333	2.333	0.87	0.85
Avg				1.55	1.87
Har				1.34	1.40
Med				1.53	1.6
SD				0.69	1.65

Table A.3: Unrolling Results for Individual Loops on 16-wide 2-cluster

Loop#	**Org II**	**Unroll uII**	**UnrollSR uII**	**Speedup**	
				Unroll vs Org	**UnrollSR vs Org**
1	9	4.25	4.25	2.12	2.12
2	14	8.25	1.13	1.70	12.4
3	15	7.25	7.25	2.07	2.07
4	2	0.75	0.75	2.67	2.67
5	2	1.25	1.25	1.6	1.6
6	2	1.25	1.25	1.6	1.6
7	2	1.25	1.25	1.6	1.6
8	5	4	4	1.25	1.25
9	2	2.75	2.5	0.73	0.8
10	7	6.42	6.42	1.09	1.09
11	2	2.75	2.5	0.73	0.8
12	2	2.75	2.5	0.73	0.8
13	2	0.75	0.75	2.67	2.67
14	2	0.75	0.75	2.67	2.67
15	2	0.75	0.75	2.67	2.67
16	21	4.75	2	4.42	10.5
17	2	2.25	2.25	0.89	0.89
18	2	0.75	0.75	2.67	2.67
19	3	0.75	0.75	4	4
20	15	7.75	4	1.94	3.75
21	11	5.25	5.25	2.09	2.09
22	28	34	34	0.82	0.82
23	34	13.75	13.75	2.47	2.47
24	8	10	10	0.8	0.8
25	32	33.5	33.5	0.96	0.96
26	21	5	2.5	4.2	8.4
27	23	21.5	5.5	1.05	4.18
28	13	9.25	7	1.41	1.86
29	3	0.75	0.75	4	4
30	5	2.75	2.5	1.82	2
31	5	2.5	2.5	2	2
32	18	4.5	1.25	4	14.4
33	5	2.5	2.5	2	2
34	8	4.25	4.25	1.88	1.88
35	2	1.125	1.125	1.78	1.78
36	3	2.5	2.5	1.2	1.2

Loop#	**Org II**	**Unroll uII**	**UnrollSR uII**	**Speedup**	
				Unroll vs Org	**UnrollSR vs Org**
37	2	1.125	1.125	1.78	1.78
38	2	1.333	1.333	1.50	1.50
39	15	5.75	5.75	2.61	2.61
40	2	2.5	2.5	0.8	0.8
41	5	4	4	1.25	1.25
42	5	4	4	1.25	1.25
43	2	2.25	2.25	0.89	0.89
44	2	2.5	2.5	0.8	0.8
45	3	2.333	2.333	1.29	1.26
46	3	2.5	2.5	1.2	1.2
47	3	0.375	0.375	8	8
48	2	2	2	1	1
49	2	2.5	2.5	0.8	0.8
50	5	2.5	2.5	2	2
51	2	2.5	2.5	0.8	0.8
52	2	2.5	2.5	0.8	0.8
53	2	2.5	2.5	0.8	0.8
54	3	1.5	1.5	2	2
55	5	4	4	1.25	1.25
56	5	4	4	1.25	1.25
57	2	1	0.5	2	4
58	2	2.5	2.5	0.8	0.8
59	9	3.5	3.5	2.57	2.57
60	5	2.5	2.5	2	2
61	5	4	4	1.25	1.25
62	2	2.5	2.5	0.8	0.8
63	5	2.5	2.5	2	2
64	2	2.5	2.5	0.8	0.8
65	5	2.5	2.5	2	2
66	2	2.5	2.5	0.8	0.8
67	8	5.25	5.25	1.52	1.52
68	16	10.33	3.33	1.55	4.80
69	3	1.5	1.5	2	2
70	3	1.667	2.333	1.80	1.29
71	2	2.5	2.5	0.8	0.8
Avg				1.79	2.38
Har				1.35	1.43
Med				1.6	1.60
SD				1.17	2.58

Table A.4: Unrolling Results for Individual Loops on 16-wide 4-cluster

Table A.5 lists the results for individual loops on TI TMS320C64x when loop unrolling or unroll-and-jam is applied.

Loop# is the loop number,

Org II is the II for the original loop,

Unrolled II is the II for the unrolled loop,

Unroll Amount is the unroll times plus one,

Unroll uII is the unit II for the unrolled loop,

Speedup is the speedup of the unrolled unit II compared with the original II.

Loop#	Org II	Unrolled II	Unroll Amount	Unroll uII	Speedup
1	3	2	2	1	3
2	12	24	2	12	1
3	6	6	2	3	2
4	1	0.5	2	0.25	4
5	3	9	6	1.5	2
6	3	9	6	1.5	2
7	3	9	6	1.5	2
8	3	3	2	1.5	2
9	3	12	8	1.5	2
10	6	24	4	6	1
11	2	12	8	1.5	1.33
12	3	12	8	1.5	2
13	1	1	2	0.5	2
14	1	1	2	0.5	2
15	1	0.5	2	0.25	4
16	3	8	2	4	0.75
17	2	2	2	1	2
18	1	0.5	2	0.25	4
19	1	1	2	0.5	2
20	4	6	2	3	1.33
21	3	2	2	1	3
22	15	15	1	15	1
23	3	3.5	2	1.75	1.71
24	7	13	2	6.5	1.07
25	17	17	2	8.5	2
26	3	13	8	1.63	1.85
27	22	22	2	11	2
29	1	4	6	0.67	1.5
30	2	9	6	1.5	1.33
31	1	1	2	0.5	2
32	7	13	2	6.5	1.08
33	1	1	2	0.5	2
34	2	1.5	2	0.75	2.67
35	3	11	6	1.83	1.64
37	3	9	6	1.5	2
38	3	9	6	1.5	2

Loop#	Org II	Unrolled II	Unroll Amount	Unroll uII	Speedup
39	2	1.5	2	0.75	2.67
40	3	12	8	1.5	2
41	3	3	2	1.5	2
42	3	3	2	1.5	2
43	1	1	2	0.5	2
44	3	12	8	1.5	2
45	3	9	6	1.5	2
46	2	3	2	1.5	1.33
47	2	4	6	0.67	3
48	3	3	1	3	1
49	3	12	8	1.5	2
50	1	1	2	0.5	2
51	2	2	1	2	1
52	3	12	8	1.5	2
53	3	12	8	1.5	2
54	3	2	2	1	3
55	3	3	2	1.5	2
56	3	3	2	1.5	2
57	2	9	6	1.5	1.33
58	2	2	1	2	1
59	2	2	2	1	2
60	1	1	2	0.5	2
61	3	3	2	1.5	2
62	2	2	1	2	1
63	1	1	2	0.5	2
64	2	2	1	2	1
65	1	1	2	0.5	2
66	2	2	1	2	1
69	2	2	1	2	1
70	3	1.5	2	0.75	4
71	3	12	8	1.5	2
Avg					1.94
Har					1.70
Med					2.00
SD					0.74

Table A.5: Unrolling Results for Individual Loops on TMS320C64x

Table A.6 reports the performance difference between the new unroll-and-jam method and Huang's method when the methods use different unroll amounts.

Loop # is the loop number,

Org II is the II for the original loop,

New is the unit II obtained by the new unroll-and-jam algorithm,

Fixed is the unit II obtained by Huang's method,

NewSpeedup is the speedup in unit II achieved by the new unroll-and-jam algorithm,

FixedSpeedup is the speedup in unit II achieved by Huang's method.

Loop#	**Org II**	**New**	**Fixed**	**NewSpeedup**	**FixedSpeedup**
8-wide 2-cluster					
22	28	28	34	1	0.82
24	8	8	10	1	0.8
48	2	2	2	1	1
51	2	2	2.5	1	0.8
58	2	2	2.5	1	0.8
62	2	2	2.5	1	0.8
64	2	2	2.5	1	0.8
66	2	2	2.5	1	0.8
69	3	3	1.5	1	2
Avg				1	0.96
Har				1	0.88
8-wide 4-cluster					
22	29	26.5	41.5	1.09	0.7
24	8	11.67	12.25	0.65	0.69
25	32	33.5	34.25	0.96	0.93
48	3	3	1	2.25	1.33
Avg				0.93	0.9
Har				0.91	0.84

Loop#	**Org II**	**New**	**Fixed**	**NewSpeedup**	**FixedSpeedup**
16-wide 2-cluster					
22	28	28	34	1	0.82
24	8	8	10	1	0.8
48	2	2	2	1	1
51	2	2	2.5	1	0.8
58	2	2	2.5	1	0.8
62	2	2	2.5	1	0.8
64	2	2	2.5	1	0.8
66	2	2	2.5	1	0.8
69	3	3	1.5	1	2
Avg				1	0.96
Har				1	0.88
16-wide 4-cluster					
8	5	4	4.25	1.25	1.18
22	28	34	41.5	0.82	0.67
24	8	10	12.25	0.8	0.65
25	32	33.5	34.25	0.96	0.93
27	23	21.5	21.75	1.07	1.06
38	2	1.33	2.25	1.5	0.89
41	5	4	4.25	1.25	1.18
42	5	4	4.25	1.25	1.18
45	3	2.33	5	1.29	0.6
48	2	2	2.25	1	0.89
51	2	2.5	2.5	0.8	0.8
55	5	4	4.25	1.25	1.18
56	5	4	4.25	1.25	1.18
58	2	2.5	2.5	0.8	0.8
61	5	4	4.25	1.25	1.18
62	2	2.5	2.5	0.8	0.8
64	2	2.5	2.5	0.8	0.8
66	2	2.5	2.5	0.8	0.8
68	16	10.33	10.25	1.55	1.56
69	3	1.5	1.5	2	2
70	3	1.67	1.75	1.8	1.71
Avg				1.16	1.05
Har				1.07	0.95

Table A.6: The New Algorithm vs. Fixed Unroll Amounts

Table A.7 lists the number of copies predicted by the new unroll-and-jam algorithm and the number of actual cross-cluster dependences collected by Memoria after unrolling and scalar replacement are applied.

Loop# is the loop number,

Pred is the number of predicted copies,

Dep is the number of intercluster dependences.

The loops marked "*" have a misprediction on the number of intercluter value copies.

Width	8 FUs				16 FUs			
Clusters	**2**		**4**		**2**		**4**	
Loop#	**Pred**	**Dep**	**Pred**	**Dep**	**Pred**	**Dep**	**Pred**	**Dep**
1	0	0	0	0	0	0	0	0
2*	0	0	1	4	0	0	1	4
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	1	1	3	3	1	1	3	3
6	1	1	3	3	1	1	3	3
7*	0	1	0	3	0	1	0	3
8	2	2	4	4	2	2	3	3
9	1	1	3	3	1	1	3	3
10*	3	2	9	6	3	2	9	6
11	1	1	3	3	1	1	3	3
12	1	1	3	3	1	1	3	3
13	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0
22*	0	0	51	54	0	0	48	24
23	0	0	0	0	0	0	0	0
24	0	0	9	9	0	0	3	3
25*	54	39	54	30	54	39	54	18
26	1	1	3	3	1	1	3	3
27	4	4	8	8	4	4	6	6
28	2	2	4	4	2	2	4	4
29	1	1	3	3	1	1	3	3
30	1	1	3	3	1	1	3	3
31	0	0	0	0	0	0	0	0
32*	1	0	3	0	1	0	1	0
33	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0
35	1	1	3	3	1	1	3	3
36	0	0	0	0	0	0	0	0

Width	**8 FUs**				**16 FUs**			
Clusters	**2**		**4**		**2**		**4**	
Loop#	**Pred**	**Dep**	**Pred**	**Dep**	**Pred**	**Dep**	**Pred**	**Dep**
37	1	1	3	3	1	1	3	3
38	1	1	3	3	1	1	2	2
39	0	0	0	0	0	0	0	0
40	1	1	3	3	1	1	3	3
41	2	2	4	4	2	2	3	3
42	2	2	4	4	2	2	3	3
43	0	0	0	0	0	0	0	0
44	1	1	3	3	1	1	3	3
45	1	1	3	3	1	1	2	2
46	0	0	0	0	0	0	0	0
47	1	1	3	3	1	1	3	3
48	0	0	0	0	0	0	0	0
49	1	1	3	3	1	1	3	3
50	0	0	0	0	0	0	0	0
51	0	0	4	4	0	0	2	2
52	1	1	3	3	1	1	3	3
53	1	1	3	3	1	1	3	3
54	0	0	0	0	0	0	0	0
55	2	2	4	4	2	2	3	3
56	2	2	4	4	2	2	3	3
57	1	1	3	3	1	1	3	3
58	0	0	4	4	0	0	2	2
59	0	0	0	0	0	0	0	0
60	0	0	0	0	0	0	0	0
61	2	2	4	4	2	2	3	3
62	0	0	4	4	0	0	2	2
63	0	0	0	0	0	0	0	0
64	0	0	4	4	0	0	2	2
65	0	0	0	0	0	0	0	0
66	0	0	4	4	0	0	2	2
67	0	0	0	0	0	0	0	0
68	2	2	4	4	2	2	3	3
69	0	0	4	4	0	0	2	2
70*	4	2	8	5	4	2	6	3
71	1	1	3	3	1	1	3	3

Table A.7: Predicted Copies vs. Intercluter Dependences

Appendix B

Individual Loop Performance for Loop Fusion

Table B.1, B.2, B.3, B.4, and B.5 presents the results for individual loops on different clustered VLIW architectures when loop fusion and/or unrolling are performed. In these tables,

Loop Set# is the number of loop set,

Org II is the II for the original loop,

Unroll uII is the unit II for the unrolled loop,

Fuse II is the II for the fused loop,

FusUnroll uII is the unit II for the fused and unrolled loop,

Fused vs Org is the speedup of the fused II compared with the sum of original IIs,

FusUnroll vs Unroll is the speedup of the unit II for the fused and unrolled loop compared with the sum of unrolled unit IIs,

FusUnroll vs Org is the speedup of the unit II for the fused and unrolled loop compared with the sum of original IIs,

Avg is the average mean speedup in unit II,

Har is the harmonic mean speedup in unit II,

Med is the median speedup in unit II,

SD is the standard deviation.

Loop Set#	**Org II**	**Unroll uII**	**Fuse II**	**FusUnroll uII**	**Speedup Fused vs Org**	**Speedup FusUnroll vs Unroll**	**Speedup FusUnroll vs Org**
1	3 3	2.33 2.5	3	2.5	2	1.93	2.4
2	5 2	3 2	6	4.5	1.17	1.11	1.56
3	3 2	3 2	2	2	2.5	2.5	2.5
4	5 5	3 4	7	5.5	1.43	1.27	1.82
5	5 2	3 2	6	3	1.17	1.67	2.33
6	5 2	3 2	6	3	1.17	1.67	2.33
7	3 3	3 1.5	3	3	2	1.5	2
8	4 4	4 4	4	4	2	2	2
9	4 4	4 4	7	7	1.14	1.14	1.14
10	2 2	2 2	3	3	1.33	1.33	1.33
11	2 2	1.67 1.67	3	2.13	1.33	1.57	1.88
12	3 3 3	2 2 2	5	5	1.8	1.2	1.8
Avg					1.59	1.57	1.92
Har					1.48	1.49	1.82
Med					1.38	1.54	1.94
SD					0.45	0.41	0.43

Table B.1: Fusion Results for Individual Loops on 8-wide 2-cluster

Loop Set#	Org II	Unroll uII	Fuse II	FuseUnroll uII	Speedup		
					Fused vs Org	**FusUnroll vs Unroll**	**FusUnroll vs Org**
1	4 4	2.5 2.25	5	2.63	1.6	1.81	3.05
2	5 3	2.5 2.5	6	5	1.33	1	1.6
3	5 3	5 3	3	3	2.67	2.67	2.67
4	5 5	2.5 4.25	7	5.67	1.43	1.19	1.76
5	5 3	2.5 2.5	6	2.67	1.33	1.88	3
6	5 3	2.5 2.5	6	2.67	1.33	1.88	3
7	3 3	1.25 1.75	3	3	2	1	2
8	4 4	4 4	4	4	2	2	2
9	4 4	4 4	7	7	1.14	1.14	1.14
10	3 3	3 3	5	3	1.2	2	2
11	3 3	2 2	3	4.5	2	0.89	1.33
12	3 3 3	3 3 3	5	6	1.8	1.5	1.5
Avg					1.65	1.58	2.09
Har					1.56	1.41	1.89
Med					1.51	1.65	2
SD					0.45	0.54	0.68

Table B.2: Fusion Results for Individual Loops on 8-wide 4-cluster

Loop Set#	Org II	Unroll uII	Fuse II	FusUnroll uII	Speedup Fused vs Org	Speedup FusUnroll vs Unroll	Speedup FusUnroll vs Org
1	3 3	2.25 2.5	3	2.5	2	1.9	2.4
2	5 2	3 2	6	4.5	1.17	1.11	1.56
3	2 2	2 2	2	2	2	2	2
4	5 5	3 4	7	5.5	1.43	1.27	1.82
5	5 2	3 2	6	3	1.17	1.67	2.33
6	5 2	3 2	6	3	1.17	1.67	2.33
7	3 3	3 1.5	3	3	2	1.5	2
8	4 4	4 4	4	4	2	2	2
9	4 4	4 4	7	7	1.14	1.14	1.14
10	2 2	2 2	2	2	2	2	2
11	2 2	1.25 1.25	2	2.17	2	1.15	1.85
12	3 3 3	2 2 2	5	5	1.8	1.2	1.8
Avg					1.66	1.55	1.94
Har					1.56	1.47	1.86
Med					1.9	1.58	2
SD					0.4	0.37	0.35

Table B.3: Fusion Results for Individual Loops on 16-wide 2-cluster

Loop Set#	**Org II**	**Unroll uII**	**Fuse II**	**FusUnroll uII**	**Speedup**		
					Fused vs Org	**FusUnroll vs Unroll**	**FusUnroll vs Org**
1	3 3	2.33 2.5	3	2.5	2	1.93	2.4
2	5 2	2.5 2.5	6	4.67	1.17	1.07	1.5
3	3 2	3 2	2	2	2.5	2.5	2.5
4	5 5	2.5 4	7	5.67	1.43	1.15	1.76
5	5 2	2.5 2.5	6	2.67	1.17	1.88	2.63
6	5 2	2.5 2.5	6	2.67	1.17	1.88	2.63
7	3 3	1.5 1.67	3	3	2	1.06	2
8	4 4	4 4	4	4	2	2	2
9	4 4	4 4	7	7	1.14	1.14	1.14
10	2 2	2 2	3	3	1.33	1.33	1.33
11	2 2	1.13 1.33	3	4	1.33	0.61	1
12	3 3 3	2 2 2	5	5	1.8	1.2	1.8
Avg					1.59	1.48	1.89
Har					1.48	1.29	1.72
Med					1.38	1.27	1.9
SD					0.45	0.54	0.57

Table B.4: Fusion Results for Individual Loops on 16-wide 4-cluster

Loop Set#	**Org II**	**Fuse II**	**Fused vs Org**
1	2.5 2	3	1.5
2	1.5 1.5	1.5	2
3	12 2	12	1.17
4	1.5 2	2	1.75
5	1.5 1.5	1.5	2
6	1.5 1.5	1.5	2
7	1.5 1.5	3	1
8	26	26	1
9	16	15	1.07
10	14	14	1
11	3 3	3	2
12	17	17	1
Avg			1.46
Har			1.33
Med			1.33
SD			0.46

Table B.5: Fusion Results for Individual Loops on TMS320C64x

Bibliography

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois, 1978.

[2] V. Adve, J-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[3] A. Aiken and A. Nicolau. A Development Environment for Horizontal Microcode. *IEEE Trans on Software Engineering*, 14(5):584–594, May 1988.

[4] A. Aletá, J. Codina, J. Sánchez, and A. González. Graph-partitioning based instruction scheduling for clustered processors. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 150–159, Austin, Texas, December 2001.

[5] V.H. Allan, R. Jones, R. Lee, and S.J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3), September 1995.

[6] F.E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[7] J.R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the Fourteenth ACM Symposium on the Principles of Programming Languages*, Munich, West Germany, January 1987.

[8] J.R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In *Supercomputers: Design and Applications*, pages 186–205. IEEE Computer Society Press, Silver Spring, MD., 1984.

[9] J.R. Allen and K. Kennedy. Vector register allocation. Technical Report TR86-45, Department of Computer Science, Rice University, 1988.

[10] R. Allen and K. Kennedy. *Optimizing Compiler for Modern Architectures*. Morgan Kaufmann, San Francisco, CA, 2002.

[11] S. Banerjia, W. A. Havanki, and T. M. Conte. Treegion scheduling for highly parallel processors. In *European Conference on Parallel Processing*, pages 1074–1078, 1997.

[12] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 275–284, Portland, OR, July 1989.

[13] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceedings of SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 53–65, June20-22 1990.

[14] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *Journal of Parallel and Distributed Computing*, 5:334–358, 1988.

[15] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for vliw's: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 292–300, Portland, OR, December 1-4 1992.

[16] S. Carr. *Memory-Hierarchy Management*. PhD thesis, Rice University, Department of Computer Science, September 1992.

[17] S. Carr. Combining optimization for cache and instruction-level parallelism. Technical Report 95-05, Michigan Technological University, September 1995.

[18] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, Maui, HI, January 1996.

[19] S. Carr and Y. Guan. Unroll-and-jam using uniformly generated sets. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park, NC, December 1997.

[20] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software Practice and Experience*, 24(1):51–77, January 1994.

[21] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[22] G. Desoli. Instruction assignment for clustered vliw dsp compilers: A new approach. Technical Report HPL-98-13, HP Laboratories Cambridge, Februray 1998.

[23] D.J. DeWitt. *A Machine-Independent Approach to the Production of Optimal Horizontal Microcode*. PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI, 1976.

[24] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 68–77, Albuquerque, NM, June 1993.

[25] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1985. PhD thesis, Yale, 1984.

[26] J. Eyre and J. Bier. The evolution of dsp processors. Technical report, Berkeley Design Technology Inc., Berkeley, CA, 2000.

[27] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[28] J.A. Fisher. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling*. PhD thesis, Courant Institute of Mathematical Sciences, New York University, New York, NY, October 1979.

[29] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[30] G. Goff, Ken Kennedy, and C-W. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language and Implementation*, volume 26, pages 15–29, Toronto, Canada, June 1991.

[31] R. Gupta and M. L. Soffa. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

[32] J. Hiser, S. Carr, P. Sweany, and S.J. Beaty. Register partitioning for software pipelining with partitioned register banks. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 211–218, Cancun, Mexico, May 2000.

[33] Jason Hiser, Steve Carr, and Philip Sweany. Global register partitioning. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compiler Techniques*, pages 13–23, Philadelphia, PA, October 2000.

[34] Xianglong Huang, Steve Carr, and Philip Sweany. Loop transformations for architectures with partitioned register banks. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, pages 48–55, Snowbird, UT, June 2001.

[35] S. Jang, S. Carr, P. Sweany, and D. Kuras. A code generation framework for VLIW architectures with partitioned register files. In *Proceedings of the Third International Conference on Massively Parallel Computing Systems (MPCS)*, pages 61–69, April 1998.

[36] N.P. Jouppi and D.W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 272–282, Boston, MA, April 1989. Comparison and evaluation of two methods.

[37] K. Kennedy. Fast greedy weighted fusion. In *Proceedings of the 2000 International Conference on Supercomputer*, May 2000.

[38] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[39] K. Kennedy and K. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report 93-208, Rice University, Aug 1993.

[40] D. Kuck. *The Structure of Computers and Computations Volume 1.* John Wiley and Sons, New York, 1978.

[41] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eight ACM Symposium on the Principles of Programming Languages*, 1981.

[42] D. Kuck, Y. Muraoka, and S.Chen. On the number of operations simultaneously executable in fortran-like program and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[43] D. Kuras, S. Carr, and P. Sweany. Value cloning for architectures with partitioned register banks. In *Proceedings of the 1998 Worshop on Compiler and Architecture Support for Embedded Systems*, Washington D.C., December 1998.

[44] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of ISCA-19*, pages 46–57, Gold Coast, Australia, May 1992.

[45] M.S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[46] L.M. Lavery and W.W. Hwu. Unrolling-based optimization for modulo scheduling. Technical report, University of Illinois, 1995.

[47] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

[48] E. Nystrom and A.E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*, pages 103–114, Dallas, TX, December 1998.

[49] E. Özer, S. Banerjia, and T.M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*, pages 308–316, Dallas, TX, December 1998.

[50] D. A. Padua. *Multiprocessor: Discussion of Some Theoretical and Practical Problems*. PhD thesis, University of Illinois at Urbana Champaign, 1979.

[51] D.A. Poplawski. The unlimited resource machine (URM). Technical Report 95-01, Michigan Technological University, January 1995.

[52] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelined loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, November 29-Dec 2 1994.

[53] B. R. Rau and J. A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[54] Jesùs Sànchez and Antonio Gonzàlez. The effectiveness of loop unrolling for modulo scheduling in clustered VLIW architectures. In *Proceedings of the 2000 International Conference on Parallel Processing*, Toronto, Canada, August 2000.

[55] Jesùs Sànchez and Antonio Gonzàlez. Instruction scheduling for clustered VLIW architectures. In *Proceedings of 13th International Symposium on System Systhesis (ISSS-13)*, Madrid, Spain, September 2000.

[56] Vivek Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 153–166, Sante Fe, NM, May 2000.

[57] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures : A design space approach*. Addison-Wesley, New York, NY, 1997.

[58] P. Sweany and S. Beaty. Dominator-path scheduling — a global scheduling method. In *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*, pages 260–263, Portland, OR, December 1992.

[59] Philip H. Sweany and Steven J. Beaty. Overview of the Rocket retargetable C compiler. Technical Report CS-94-01, Department of Computer Science, Michigan Technological University, Houghton, January 1994.

[60] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, 2000. literature number SPRU189.

[61] Texas Instruments. *TMS320C6000 Optimizing Compiler User's Guide*, 2000. literature number SPRU187.

[62] G.S. Tjaden and M.J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, C-19(10):889–895, Oct 1970.

[63] J. Warren. A Hierarchical Basis for Reordering Transformations. In *Conference Record of the 11th ACM Symposium on the Principles of Programming Languages*, pages 272–282, January 1984.

[64] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[65] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, October 1982. Parallelization.

[66] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Modulo scheduling with integrated register spilling for clustered VLIW architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 160–169, Austin, Texas, December 2001.