The Pennsylvania State University

The Graduate School

College of Engineering

PROGRAM ALLOCATION AND IMPLEMENTATION
OF CACHE IN A
DATAFLOW ENVIRONMENT

A Thesis in
Computer Engineering

by

Joford T. Lim

© 2001 Joford T. Lim

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2001
We approve the thesis of Joford T. Lim.

Ali R. Hurson  
Professor of Computer Science and Engineering  
Thesis Advisor  
Chair of Committee

Mary Jane Irwin  
Distinguished Professor of Computer Science and Engineering

John J. Metzner  
Professor of Computer Science and Engineering  
and Electrical Engineering

Octavia I. Camps  
Associate Professor of Computer Science and Engineering  
and Electrical Engineering

Ali Borhan  
Professor of Chemical Engineering

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering
ABSTRACT

The success of multithreaded systems depends on how quickly context switching between threads can be achieved. Fast context switch is only possible if threads are resident in fast but small memories (such as instruction buffers, caches and registers). This however, limits the number of active threads and thus the amount of latency that can be tolerated. The generality of dataflow scheduling makes it difficult to fetch and execute a sequence of logically related sets of threads through the processor pipeline, thereby removing any opportunity to use registers across thread boundaries. Relegating the responsibilities of scheduling and storage management to the compiler alleviates this problem to some extent. In conventional architectures, the reduction in memory latencies is achieved by providing (explicit) programmable registers and (implicit) high-speed caches. Amalgamating the idea of caches or register-caches within the dataflow framework can result in a higher exploitation of parallelism and hardware utilization. This thesis investigates the suitability of cache memory in a dataflow paradigm.

We present two heuristic schemes that allow the detection, exploitation, and enhancement of temporal and spatial localities in a dataflow graph (dataflow program). Dataflow graphs are partitioned into subgraphs while preserving localities, and subgraphs are distributed among the processors in order to reduce cache misses and communication cost. The Staggered Distribution scheme was proposed to address the issue of detection and allocation of dynamic parallelism in a program. On the other hand, the Vertically Layered (VL) allocation scheme, can effectively detect spatial localities in a dataflow graph by assigning nodes connected serially to a partition (a vertical layer).

As part of this thesis, we intend to incorporate and utilize these two schemes for detecting and enhancing localities in a dataflow graph, which is essential in optimizing the effectiveness of cache in multithreaded dataflow architectures. In addition, since the Staggered scheme produces an unbalanced load among processors. An extension to the Staggered scheme that produces a more balanced distribution of iterations among processors will also be developed.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>xi</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>xi</td>
</tr>
</tbody>
</table>

## CHAPTER 1. INTRODUCTION

1. Motivation                                                                                     | 1    |
2. Thesis Objectives and Contents                                                                 | 6    |

## CHAPTER 2. CACHE IN A DATAFLOW ENVIRONMENT

2.1. Overview                                                                                     | 8    |
2.2. Dataflow Computation and Architectures                                                       | 8    |
2.3. Cache in a Dataflow Environment                                                             | 22   |
2.3.1. Locality in Dataflow Programs                                                             | 22   |
2.3.2. Limits of Dataflow Multiprocessing                                                        | 23   |
2.3.3. Cache Memory Designs with ETS                                                            | 25   |
2.3.3.1. Instruction Cache in ETS                                                               | 26   |
2.3.3.2. Operand Cache in ETS                                                                 | 29   |
2.4. Summary                                                                                     | 32   |

## CHAPTER 3. PROGRAM PARTITIONING AND ALLOCATION

3.1. Overview                                                                                     | 34   |
3.2. Partitioning Programs into Threads                                                           | 35   |
3.3. Allocation Problem ........................................................................................................ 37
   3.3.1. Program Allocation on Dataflow Machines ....................................................... 37
   3.3.2. Vertically Layered Allocation Scheme ............................................................... 40
3.4. Loop Allocation ................................................................................................................ 49
   3.4.1. Loop Scheduling Algorithms For DOALL Loops ............................................. 50
      3.4.1.1. Self-scheduling ..................................................................................... 52
      3.4.1.2. Fixed-size chunking ............................................................................ 52
      3.4.1.3. Guided Self-scheduling ...................................................................... 53
      3.4.1.4. Factoring ............................................................................................. 54
      3.4.1.5. Trapezoid Self-scheduling .................................................................. 54
   3.4.2. Comparative Analysis Of DOALL Loop Scheduling Schemes ......................... 55
   3.4.3. DOACROSS Loop Scheduling ........................................................................ 57
      3.4.3.1. The Regular DOACROSS Model ....................................................... 59
      3.4.3.2. Pre-synchronized Scheduling (PSS) .................................................... 62
      3.4.3.3. Staggered Distribution Scheme ........................................................... 63
   3.4.4. Comparison of DOACROSS Scheduling Schemes .......................................... 65
   3.4.5. Performance Results of Loop Scheduling Schemes ........................................... 68
3.5. Summary ...................................................................................................................... 75

CHAPTER 4. CYCLIC STAGGERED .................................................................................. 78
   4.1. Overview .................................................................................................................. 78
   4.2. Cyclic Staggered Distribution (CSD) ..................................................................... 78
   4.3. Performance of Cyclic Staggered Distribution .................................................... 79
   4.4. Summary ................................................................................................................ 82
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Dataflow graph</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Conditional graph</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Loop graph</td>
<td>13</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>The general organization of the dynamic dataflow model</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>An organization of a pure-dataflow processing element</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>An organization of a hybrid processing element</td>
<td>19</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>An organization of a macro-dataflow processing element</td>
<td>20</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>ETS representation of a dataflow program execution</td>
<td>27</td>
</tr>
<tr>
<td>Figure 2.9</td>
<td>Instruction Cache Organization</td>
<td>28</td>
</tr>
<tr>
<td>Figure 2.10</td>
<td>Operand Cache Organization</td>
<td>30</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>An example of a dataflow graph</td>
<td>41</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Vertically layered graph of Figure 3.1</td>
<td>44</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Type A inter-PE communication behavior</td>
<td>46</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Type B inter-PE communication behavior</td>
<td>48</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Allocation of a DOACROSS Loop</td>
<td>61</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Performance of DOALL scheduling algorithms on matrix multiplication</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>(N = 300)</td>
<td></td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Performance of DOALL scheduling algorithms on adjoint convolution</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>(N = 100)</td>
<td></td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Maximum speedup (MS), n = 2000, C/T = 0.5</td>
<td>72</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Number of PEs to attain maximum speedup for Cyclic scheduling</td>
<td>74</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Speedup for loop 13 - nCUBE 2 system</td>
<td>76</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.1</td>
<td>Comparative analysis of the staggered schemes, $C/T = 3.0$</td>
<td>83</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparative analysis of the staggered schemes, $C/T = 5.0$</td>
<td>84</td>
</tr>
<tr>
<td>4.3</td>
<td>Speedup of the staggered schemes for Loop 13 - nCUBE 2 system</td>
<td>85</td>
</tr>
<tr>
<td>5.1</td>
<td>Speedup for Livermore Loop 19, $C/E = 10$</td>
<td>90</td>
</tr>
<tr>
<td>5.2</td>
<td>Speedup for Livermore Loop 19, $C/E = 50$</td>
<td>91</td>
</tr>
<tr>
<td>5.3</td>
<td>Speedup for Simple, $C/E = 10$</td>
<td>92</td>
</tr>
<tr>
<td>5.4</td>
<td>Speedup for Simple, $C/E = 50$</td>
<td>93</td>
</tr>
<tr>
<td>5.5</td>
<td>Speedup for Livermore Loop 19, $C/E = 10$</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>Speedup for Livermore Loop 19, $C/E = 50$</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>Speedup for Simple, $C/E = 10$</td>
<td></td>
</tr>
<tr>
<td>5.8</td>
<td>Speedup for Simple, $C/E = 50$</td>
<td></td>
</tr>
<tr>
<td>5.9</td>
<td>An example of threads assigned to a vertical layer</td>
<td>97</td>
</tr>
<tr>
<td>5.10</td>
<td>Memory allocation of Figure 6.1</td>
<td>98</td>
</tr>
<tr>
<td>5.11</td>
<td>Effect of different values of $x$ on instruction cache for Sloop74</td>
<td>101</td>
</tr>
<tr>
<td>5.12</td>
<td>Effect of different values of $x$ on operand cache for Sloop74</td>
<td>103</td>
</tr>
<tr>
<td>5.13</td>
<td>Performance of instruction cache using VL-cache and E-level ordering for Sloop74</td>
<td>104</td>
</tr>
<tr>
<td>5.14</td>
<td>Performance of operand cache using VL-cache and E-level ordering for Sloop74</td>
<td>105</td>
</tr>
<tr>
<td>5.15</td>
<td>Performance of instruction cache using VL-cache and E-level ordering for FFT</td>
<td>106</td>
</tr>
<tr>
<td>5.16</td>
<td>Performance of operand cache using VL-cache and E-level ordering for FFT</td>
<td>107</td>
</tr>
<tr>
<td>5.17</td>
<td>Performance of instruction cache using VL-cache and E-level ordering for Simple</td>
<td>108</td>
</tr>
<tr>
<td>5.18</td>
<td>Performance of operand cache using VL-cache and E-level ordering for Simple</td>
<td>109</td>
</tr>
<tr>
<td>5.19</td>
<td>Percentage improvement of VL-cache over E-level ordering for FFT</td>
<td>110</td>
</tr>
<tr>
<td>5.20</td>
<td>Percentage improvement of VL-cache over E-level ordering for Simple</td>
<td>111</td>
</tr>
<tr>
<td>5.21</td>
<td>Performance of operand cache with prefetching, using VL-cache and E-level ordering for Sloop74. Operand cache size = 2K bytes</td>
<td>113</td>
</tr>
<tr>
<td>5.22</td>
<td>Performance of operand cache with prefetching, using VL-cache and E-level ordering for Sloop74. Operand cache size = 8K bytes</td>
<td>114</td>
</tr>
<tr>
<td>5.23</td>
<td>Performance of operand cache with prefetching, using VL-cache and E-level ordering for Simple</td>
<td></td>
</tr>
</tbody>
</table>
E-level ordering for Sloop74. Operand cache size = 32K bytes

Figure 6.16. Performance of operand cache with prefetching, using VL-cache for Simple

Figure 6.17. Performance of operand cache with prefetching, using VL-cache for FFT
LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Comparative analysis of DOALL scheduling algorithms</td>
<td>56</td>
</tr>
<tr>
<td>3.2</td>
<td>Number of iterations assigned to a processor at each scheduling step</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>with $n = 1000$, $P = 4$</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Comparison of DOACROSS scheduling algorithms</td>
<td>66</td>
</tr>
<tr>
<td>3.4</td>
<td>Number of iterations assigned to a processor at each scheduling step</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>with $T = 10$, $d = 2$, $n = 500$, $C = 5$, $P = 4$</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Speedup of Staggered Distribution relative to Static Chunking Su (SC) and Cyclic Scheduling Su(CYC) for the Livermore Loops with $C/E = 30$</td>
<td>75</td>
</tr>
<tr>
<td>4.1</td>
<td>Comparison of DOACROSS scheduling algorithms</td>
<td>80</td>
</tr>
<tr>
<td>4.2</td>
<td>Number of iterations assigned to a processor at each scheduling step</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>with $T = 10$, $d = 2$, $n = 500$, $C = 5$, $P = 4$</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Program Characteristics</td>
<td>100</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to express my gratitude to my thesis advisor, Dr. Ali R. Hurson, for his encouragement, patience, guidance, and support throughout the preparation of this thesis. I would also like to thank the committee members, Dr. Mary Jane Irwin, Dr. John J. Metzner, Dr. Octavia I. Camps, and Dr. Ali Borhan.

I wish to thank Dr. Krishna Kavi at The University of Northern Texas for providing the facility to conduct my research. Special thanks go to Finny Samuel and Drew Meyer of the University of Texas at Arlington.

I would like to acknowledge the National Science Foundation, which supported part of this work under Grant MIP-9622836.

Finally, I would like to thank my wife, Irene, for her patience, support, and most of all, for always believing in me.
CHAPTER 1
INTRODUCTION

1.1. Motivation

The traditional approaches to concurrent processing are based on the control-flow model of computation, where a program counter is used to thread the execution of a task. In a multiprocessor organization, the basis of the control-flow model is extended to allow more than one thread of control to be active at an instance. This is achieved by special control operators as a means for activating and synchronizing these threads.

Architects of such an organization must address the loss in processor efficiency due to two fundamental issues: memory latencies and synchronization overhead [12]. Memory latency is the time elapsed between issuing a memory request and receiving the corresponding response. If a referenced data item is situated in a non-local memory module, the access time is increased by the transit time through the communication network. This time is typically orders of magnitude longer than the instruction execution cycle [30]. Synchronization, the second fundamental issue in designing multiprocessors, is the need to enforce the ordering of instruction execution according to their data dependencies. In conventional multiprocessors, this is achieved by providing synchronization primitives in the program. The main problem is the relatively high overhead resulting from most existing synchronization primitives.

Context switching during a non-local memory access can be used to avoid the processor idle time caused by memory latency. Unfortunately, most conventional multiprocessors incur a very large overhead during a context switch. Another approach to reduce memory latency is through the use of global caching, i.e., caching global data in a processors' local cache. The biggest problem with global caching is keeping the caches coherent [4]. If two processors have a copy of a cache line, and one modifies that line, the other processor must somehow be automatically notified of the change. However, cache coherency protocols in a distributed memory machine are very complicated [4]. Also, support for global coherent cached memory usually requires a fairly sizable investment in hardware.
When an instruction requires a synchronization event to occur before it continues execution, the instruction can either wait for that event, which is achieved by idling the processor or perform a context switch. The first choice, while simple to implement, wastes precious computing resources if the wait is long. Not releasing the processor while waiting for a synchronization event can also cause deadlock. The second carries the overhead associated with context switching. As a consequence, parallelism is usually exploited at a large grain level, thus sacrificing the potential fine grain parallelism that might exist in an algorithm.

The dataflow model of computation was proposed as an alternative to the conventional control-flow model of computation. It explicitly addresses the issue of programmability as well as memory latency and synchronization. Programmability is facilitated through high-level declarative or functional languages, which obey the single-assignment principle [6]. The dataflow model of computation is different from the conventional control-flow model in that dataflow operations are asynchronous, i.e., the execution of an instruction is based on the availability of its operands, rather than being driven by a program counter. Therefore, instructions in the dataflow model do not impose any constraints on sequencing except the data dependencies in the program. Hence, the dataflow graph representation of a program exposes all forms of parallelism eliminating the need to explicitly manage parallel execution of a program. Theoretically, in a dataflow machine, maximal concurrency can be exploited, constrained only by the availability of hardware resources. Dataflow machines may be regarded as an extreme example of machines with multiple threads, in which each instruction constitutes an independent thread, and only non-suspended threads are scheduled to be executed. Since instruction execution is triggered by the availability of operands, the computation is capable of tolerating arbitrary memory latencies. Furthermore, synchronization is enforced at the instruction level, because every instruction waits for all its operands to be produced before execution. Dataflow machines treat each instruction as a task, and by applying a small synchronization cost, offer the ultimate flexibility in scheduling instructions.

The dataflow model of computation has been a subject of study for over thirty years. Basic studies of dataflow computing have been promoted by various research groups and a number of hardware prototypes have been built and evaluated [35, 72, 102]. Studies from past dataflow projects, however, have revealed a number of inefficiencies in dataflow computing. For example: i) the dataflow model incurs
more overhead in the execution of an instruction compared to its control-flow counterpart due to its fine-grained approach to parallelism, ii) the overhead involved in the detection of enabled instructions and the construction of result tokens generally result in poor performance in applications with low-degree of parallelism, iii) the task of managing resources, generally, is a complicated process, and finally, iv) the representation and handling of large data structures poses serious problems.

In spite of these shortcomings, a renewed interest has emerged in the area of dataflow computing. This revival was partially due to: i) the lack of developments in the conventional parallel processing arena, ii) the lessons learned from previous attempts in building dataflow machines and the realization of the need for relaxing control-flow dependencies to achieve higher parallelism, iii) a change in the viewpoint on the actual concept of dataflow and its implementation, and iv) the viewpoint that the dataflow concept should be supported by appropriate compilation and program representation schemes rather than with specific hardware support. These changes allow the incorporation of conventional control-flow methods into the dataflow approach, thus alleviating the inefficiencies associated with the pure dataflow method. This new computational model, multithreaded dataflow, and our past experiences have made us to believe that many control-flow issues can be easily studied in a dataflow environment with a goal to improve its computational power.

Conventional compilers for control-flow multiprocessors attempt to extract parallelism embedded in a sequential program. This is a difficult if not impractical task and will not totally extract all the parallelism in a program. In fact, some dependencies that restrict the extraction of parallelism are due to reuse of memory locations. At this point in time, literature has not reported of any compiler that takes standard sequential languages and compiles them for commercial parallel machines with any success. The results published in [15], for example, shows that poor speedup is achieved for most of the programs in the Perfect Benchmarks suite when a parallelizing compiler is used for the Alliant/FX80 system. As a result, every vendor has its version of parallel extensions to Fortran and C which allows a user to express his/her algorithm for a particular architecture [39].

Dataflow machines on the other hand, use implicit parallel languages such as Id, in which the compiler merely exposes the parallelism implicit in the program, regardless of the physical characteristics of the underlying machine architecture [39]. In fact, to a degree, the programmer does not even have to be
aware of the parallelism in the program. Instead of the conventional approach of extracting parallelism from a sequential program, multithreaded dataflow architectures start with a dataflow language (Id) compiled to dataflow graphs, which exposes all the available parallelism in the program and partitions the graph into threads to try to constrain this parallelism.

The success of multithreaded dataflow however, depends on how quickly context switching can be supported. This is only possible if threads are resident in fast memories, such as cache. The sizes of cache are usually small, hence the number of active threads and thus the amount of latency that can be tolerated is limited. The generality of dataflow scheduling makes it difficult to execute a logically related set of threads through the processor pipeline, thereby removing any opportunity to utilize registers across thread boundaries. Relegating the responsibilities of scheduling and storage management to the compiler alleviates this problem to some extent. Appropriate means of directing scheduling based on some global-level understanding of program execution will be crucial to the success of future dataflow architectures [59].

The application of cache in a dataflow environment is a phenomenon that attempts to utilize the successful artifacts from control-flow environments in a dataflow context. Some work that has been conducted with this regard include the use of operand (OM) and instruction (IM) cache memories in Dataflow Machine-II (DFM-II) [94], and register caches in Super-Actor Machine (SAM) [40]. The issues related to the use of cache memories in dataflow machines have also been investigated by using an abstract machine based on a single ETS processor model of execution [53, 75, 89]. It was found that instruction cache memory design issues are very similar to those in control-flow architectures. Designers of *T have also studied the issue of cache memory within the scope of the *T-NG — Next Generation [4].

The aforementioned efforts have shown the feasibility of the application of cache within the scope of a dataflow environment. Despite the progress that has been reported, however, the literature has been silent about the exploitation of localities within a dataflow programming environment. Specially the application of different optimization techniques as a means to enhance localities in a dataflow program has not been fully addressed and evaluated. The main goal of this research is to study, develop, and analyze optimization techniques which detect and enhance localities in a dataflow program with an eye to study the application of cache memory in a dataflow environment.
To exploit the effectiveness of cache, a scheme that detects localities in a dataflow graph, partitions the graph into subgraphs while preserving the localities, and distributes the subgraphs among the processors in a multiprocessor environment is needed. An equally effective cache placement/replacement and fetching/prefetching policy is also required to ensure that instructions and data needed by a thread is resident in the cache before the thread is activated.

In a sequential environment, the instructions of a loop are reused in successive iterations. If instructions are similarly reused in a dataflow environment, temporal locality will exist. Straight-line code may also produce spatial locality in a dataflow environment. An exploitable spatial locality is a set of instructions which would constitute a spatial locality if they were grouped together in the virtual address space, and is represented by a path of activity determined by the data dependencies in the program — i.e., a dataflow graph is partitioned into a collection of vertical layers, where nodes in each layer are serially dependent on each other.

The Staggered Distribution scheme [47, 65] was proposed to address the issue of detection and allocation of dynamic parallelism in a program. In this scheme, temporal and spatial locality are preserved by grouping together logically related threads (instructions) belonging to the same iteration and successive iterations. On the other hand, the Vertically Layered (VL) allocation scheme proposed in [61], can effectively detect spatial localities in a dataflow graph by assigning nodes connected serially to a partition (a vertical layer). Similar to the staggered distribution scheme, the VL allocation scheme takes into account the effect of inter-processor communication delays in determining its partitions.

As part of this thesis, we intend to incorporate and utilize these two schemes for detecting and enhancing localities in a dataflow graph, which is essential in optimizing the effectiveness of cache in multithreaded dataflow architectures. In conjunction, we will also be investigating different placement/replacement and fetching/prefetching policies appropriate for caches in a multithreaded dataflow environment.

1.2. Thesis Objectives and Contents
The objectives of this research are twofold: i) incorporating the Staggered Distribution scheme with the VL allocation scheme to develop a partitioning scheme that preserves localities, reduces the effect of inter-processor communication delays, and optimizes the effectiveness of cache in multithreaded dataflow architectures. This would involve addition of statistical information for each vertical layer to aid the run-time system in implementing frame allocation, prefetching, and placement/replacement policies. The partitions produced by this new scheme will be executed on a multithreaded dataflow simulator with cache. Performance results will be analyzed to determine the effectiveness of this new scheme and to determine what further modifications are necessary, and ii) investigating different placement/replacement and fetching/prefetching policies appropriate for caches in a multithreaded dataflow environment. This would be conducted in conjunction with the first objective. Various prefetching and placement/replacement policies would be considered. The results will again be analyzed to determine the necessary actions to take. In addition, thread scheduling or process control mechanisms will also be developed to alleviate the effect of limited cache size, which effectively reduces the amount of latency that can be tolerated by a processor.

The rest of the thesis is organized as follows: Chapter 2 briefly overviews the concept of dataflow computation. Moreover, it discusses the issues affecting the incorporation of cache in a dataflow environment. It also covers how detection of locality can enhance the performance of cache memories. Chapter 3 presents some methods that can be used to enhance the locality in a dataflow program. We will show how the Vertically Layered Allocation Scheme and the Staggered Distribution Scheme can effectively detect both spatial and temporal localities in a dataflow graph. A survey of loop scheduling schemes and their importance is also discussed. A modified version of the Staggered Distribution Scheme called Cyclic Staggered is introduced in Chapter 4. The performance improvement of this new version will be examined as well. In Chapter 5, the Vertically Layered Allocation Scheme and the two Staggered Schemes are combined to produce the VL-Stag Allocation Scheme. This new scheme is intended to address the shortcomings of the Vertically Layered Allocation Scheme. Performance results of this new scheme are also presented. A new locality enhancing policy that improves the performance of cache memory is proposed in Chapter 6. This new policy utilizes the partitions produced by the VL-Stag Allocation Scheme and reorders the instructions in the program to enhance locality. Fetching/prefetching
policies that complement this new locality enhancing policy will also be introduced. The effectiveness of this new policy is studied through simulation. Finally, we provide a brief conclusion and directions for future developments in Chapter 7.
CHAPTER 2
CACHE IN A DATAFLOW ENVIRONMENT

2.1. Overview

The incorporation of conventional control-flow method of computation into the dataflow paradigm (multithreading) has injected new interest in investigating issues that pertain to control-flow and how these issues can be applied in a dataflow environment. One particular issue that is of interest is the incorporation of cache in a dataflow environment. Since cache has proven its effectiveness when localities govern the structure of the underlying programs, it would be interesting to investigate the presence of localities in dataflow programs and how cache can be incorporated into dataflow architectures.

In this chapter, we will first discuss the dataflow concept and its architecture, then we will look into some issues that can affect the performance of cache in a dataflow environment. We will also look into how these issues can be handled by presenting some methods that have been developed. Finally, some important issues which have not been emphasized in the literature and have a direct link to this work will be discussed.

2.2. Dataflow Computation and Architectures

In the dataflow model of computation, a program is described as a directed graph in which nodes represent instructions and the arcs represent paths along which data values flow. The data values are carried in packets known as tokens. Execution (firing) of each instruction is triggered when input data tokens are present on all of its input arcs. Execution may be immediate or delayed, depending on the availability of processing resources. The input data tokens, which are said to match together, are extracted from the input arcs, the operation is performed, and the result data token(s) are generated and sent to the destination node(s) along the output arc(s).
The dataflow model possesses several interesting properties. It does not impose spurious dependencies on the order of execution of instructions. Dataflow instructions are functional (free from side-effects) and their execution are triggered by input data token availability. Hence, execution of a dataflow instruction cannot be delayed by conflicts of access to memory. Totally decentralized instruction control is thus possible, because scheduling of instruction execution does not rely on a global state. Finally, with the fine-grain granularity used, each instruction constitutes a logical process, and so the number of instantaneously active processes can be large.

As an example, Figure 2.1 shows the dataflow program graph for the following expression.

Let \( x = a \ast b \);
\( y = 4 \ast c \)
in \( (x + y) \ast (x - y)/c \).

Any arithmetic or logical expression can be translated into a dataflow program graph in a straightforward manner. When a node fires, a data token is removed from each input arc, a result is computed using these data values, and a token containing the result is produced on each output arc.

Nodes \( s1 \) and \( s2 \) in Figure 2.1 are both enabled for execution as soon as tokens are placed on the input arcs \( a, b, \) and \( c \). They may fire simultaneously, or in any order. The results are the same in either case. The result of an operation is purely a function of the input values; there are no implicit interactions between nodes via side effects (e.g., through shared memory). This example illustrates two key properties of the dataflow approach; (a) parallelism, i.e., nodes may potentially execute in parallel unless there is an explicit data dependence between them, and (b) determinacy, i.e., results do not depend on the relative order in which potentially parallel nodes are executed. Furthermore, notice that by supplying several sets of input tokens, distinct computations can be pipelined through the graph. In this example, a single wave of tokens on the input arcs produces a single wave of tokens on the output arcs. Graphs that have this property are called well-behaved [7]. All acyclic graphs for arithmetic and logical expressions are well-behaved.
Figure 2.1. Dataflow graph
In order to build conditional and loop program graphs, two control operators, switch and merge are introduced. Consider first the conditional graph in Figure 2.2 that represents the expression if $x < y$ then $x+y$ else $x-y$. The initial tokens provide the data input to the switches as well as input to the predicate node. The predicate node yields a single boolean value that supplies the control input to all the switches and merges. A switch routes its data input to the output arc on the True side or False side, according to the value of the control input. Thus, the wave of input tokens is directed to the True or the False arm of the merge node. As long as the arms of the conditional graph are well-behaved graphs, a single wave of tokens will eventually arrive at the data input of the appropriate side of the merge. The merge selects an input token from the True or the False side input arc, according to the value of the control input, and reproduces the data input token on the output arc. To see that the conditional graph behaves appropriately when waves of inputs are presented to it, consider the case in which the first wave of input tokens is switched to the True side, the second wave to the False side, and the tokens on the False side of the merge arrive before the tokens on the True side. The sequence of control tokens at the merge restores the proper order among the tokens on the output arcs.

The loop graph shown in Figure 2.3 computes:

$$\sum_{i=1}^{N} F(i)$$

The output of the predicate is connected to each of the switches and merges, and the graph corresponding to function $F$ is indicated by the "box" containing $F$. The initial values of $i$ and $sum$ enter the loop from the False sides of the merges, and provide data to the predicate and switches. If the predicate evaluates to True, the data values are routed to the loop body. Assuming the body is a well-behaved graph, eventually a single wave of results is produced that provides tokens on the True side of the merges. In this way, values circulate through the loop until the predicate turns to False, which causes the final values to be routed out of the loop and restores the initial false values on the control inputs to the merges. Note that if many waves of inputs are provided, only one wave at a time is allowed to enter the loop, the second wave enters the loop as soon as the first wave completes, and so on. Also note that the loop values need not
Figure 2.2. Conditional graph.
sum = 0
i = 1

Figure 2.3. Loop graph
circulate in clearly defined waves. Suppose \( F \) is a very large graph or does not fire for a long time. The index variable \( i \) may continue to circulate, causing many computations of \( F \) to be initiated. This behavior is normally referred to as dynamic unfolding of a loop.

Dataflow graphs exhibit two kinds of parallelism in instruction execution [7], i.e., spatial and temporal (dynamic) parallelism. In the first case, any two nodes can potentially be executed concurrently if there is no data dependence between them. The second form of parallelism results from pipelining independent waves of computation through the graph, i.e., execute several instances of the same node concurrently.

In the abstract dataflow model just discussed, data values are carried by tokens. These tokens travel along the arcs connecting various instructions in the program graph. The arcs are assumed to be FIFO queues of unbounded capacity. However, a direct implementation of this model has proven to be impractical. Instead the dataflow model has been classified as either static or dynamic. In the static dataflow model, a dataflow actor can be executed only when all of the tokens are available on its input arcs and no tokens exist on any of its output arc. Hence, it allows at most one instance of a node to be enabled for firing. On the other hand, the dynamic dataflow model permits activation of several instances of a node at the same time during run-time. To distinguish between different instances of a node, a tag is associated with each token that identifies the context in which a particular token is generated. An actor is considered executable when its input arcs contain a set of tokens with identical tags.

Due to dynamic dataflow model's major advantage over static — higher performance by allowing multiple tokens to exist on an arc — current dataflow research efforts indicate a trend towards adopting the dynamic dataflow model. The dynamic dataflow model was proposed by Arvind at MIT [13] and by Gurd and Watson at the University of Manchester [37]. Figure 2.4 shows the general organization of the dynamic dataflow model. Tokens are received by the matching unit, which is a memory containing a pool of waiting tokens. The unit's basic operation brings together tokens with identical tags. If a match exists, the corresponding token is extracted from the matching unit, and the matched token set is passed to the fetch unit. If no match is found, the token is stored in the matching unit to await its partner. In the fetch unit, the tags of the token pair uniquely identify an instruction to be fetched from the program memory.
Figure 2.4. The general organization of the dynamic dataflow model.
The instruction and the token pair form the enabled instruction, which is sent to the processing unit. The processing unit executes the enabled instructions and produces result tokens to be sent to the matching unit via the token queue.

Despite the dynamic dataflow model's potential for large-scale parallel computer systems, experience has identified a number of difficulties in its implementation [59]:

1. Overhead involved in matching tokens is heavy.
2. Resource allocation is a complicated process.
3. The long dataflow instruction cycle, and
4. Handling data structures is not trivial.

Since the mid-1980s, the shortcoming of the so-called conventional concurrent systems, coupled with new progress made in the dataflow arena, have motivated a new surge of interest in dataflow processing. As reported in the literature, several projects have already been established to prototype and fabricate high-performance multiprocessors dataflow machines. The success of these projects is primarily due to the shift in attitude regarding the level of parallelism to be exploited by the architecture. This implies the application of both dataflow and control-flow concepts, and development of software supports for dataflow in particular. These newer dataflow machines can be classified into three categories: pure-dataflow, macro-dataflow, and hybrid-dataflow organization.

The pure-dataflow organization is a slight modification of a dynamic dataflow machine that implements the traditional dataflow instruction cycle. Figure 2.5 shows a typical processing element (PE) based on the pure-dataflow organization. It consists of an execution pipeline connected to a token queue. The reversal of the Instruction Fetch Unit and the Matching Unit and the introduction of frames to represent contexts are the major differences between the pure-dataflow organization and the classical dataflow model. These changes are mainly due to the implementation of a new direct matching scheme, which eliminates the expensive and complex process of associative search used in previous dynamic dataflow architectures. Monsoon [72] and Epsilon-2 [35] are examples of machines based on this organization.
Figure 2.5. An organization of a pure-dataflow processing element.
The hybrid organization is more of a radical departure from the classical dynamic architectures in the sense that tokens only carry tags, and the architecture is based on conventional control-flow sequencing (Figure 2.6). Architectures based on this organization can be viewed as von Neumann machines that have been extended to support fine-grained dataflow capability. Moreover, unlike the pure-dataflow organization where token matching is implicit in the architecture, machines based on the hybrid organization provide a limited token matching capability through special synchronization primitives. P-RISC [70], *T [71], *T-NG [4, 23], and TAM [25, 26] are examples of proposals that can be categorized as hybrid organizations.

The macro-dataflow organization, shown in Figure 2.7, can be viewed as a compromise between the two aforementioned categories. The macro-dataflow architecture can be characterized by the integration of a token-based circular pipeline and an advanced control pipeline. The basic idea is to shift from exploiting fine-grain parallelism to a more coarse grain parallelism by incorporating control-flow sequencing into the dataflow approach. EM-4 is an example of a macro-dataflow organization [102].

The progress toward the exploitation of parallelism leading to a compromise between dataflow and von Neumann alleviates the inefficiencies associated with the pure dataflow model. Interestingly, the instruction-level context switching capability combined with sequential scheduling provides a different perspective on dataflow architectures — multithreading. In the context of multithreading, a thread is a sequence of statically ordered instructions where once the first instruction is executed, the remaining instructions execute without interruption — a thread defines the basic unit of work from the dataflow model standpoint that requires synchronization only at the beginning of a thread. The evolution from pure self-scheduling paradigm to multithreading requires locality and improved processor efficiency during remote memory accesses. Multithreaded dataflow architectures can be viewed as either an evolution of dataflow architectures in the direction of more explicit control over instruction execution order, or evolution of von Neumann machines in the direction of better support for synchronization and tolerance for long latency operations.

While the dataflow execution model can potentially uncover all forms and levels of parallelism in a program, in its traditional fine grain form it does not exploit any form of locality, thereby introducing
Figure 2.6. An organization of a hybrid processing element.
Figure 2.7. An organization of a macro-dataflow processing element.
unnecessary run-time overhead in the most expensive stage of the dataflow pipeline — the matching stage. Recent evidence indicates that the exploitation of locality in dataflow programs could have a dramatic impact on performance [67]. The current trend in the design of dataflow processors suggests a synthesis of traditional non-strict fine grain instruction execution and strict coarse grain execution order to exploit locality. While an increase in instruction granularity favors the exploitation of locality within a single execution thread, the resulting grain size may increase latency among execution threads — more than one operand would be required to activate a thread. Results of a recent experiment shows compelling evidence that a coarse grain execution outperforms a fine grain one on a significant number of numeric codes [67]. This suggest that the effect of increased instruction granularity on latency is minimal for a high percentage of the measured codes, and in large part is offset by available intrathread locality. This intrathread locality is the equivalent of the von Neumann instruction level locality. Hence, this new computational model and the existence of this locality have led us to believe that many control-flow issues, such as register files, cache memories, and instruction prefetch, can be easily studied in a dataflow environment with a goal to improve its computational power.

In conventional architectures, the reduction in memory latencies is achieved by providing (explicit) programmable registers and (implicit) high-speed caches. Amalgamating the idea of caches or register-caches within the dataflow framework could result in a higher exploitation of parallelism and resource utilization [43]. Localities in the organization of the conventional programming environment have made the concept of cache a viable means for reducing memory latencies. Hence, it will be interesting to:

1. Investigate the presence of localities in dataflow programs.
2. Develop proper compilation schemes which enhance localities in dataflow programs, and
3. Analyze performance improvement of dataflow architectures in the presence of cache memories.

In the next sections, we will discuss the issue of locality and how this could enhance the performance of cache in a dataflow environment, and how cache limits the amount of latency that is tolerated.
2.3. Cache in a Dataflow Environment

In general, the design of a cache is subject to more constraints and trade-offs than that of the main memory. Issues such as the placement/replacement policy, fetch/update policy, homogeneity, the addressing scheme, block size, and the cache bandwidth are among those which must be taken into consideration. Optimizing the design of a cache memory generally has four major aspects:

1. Maximizing the probability of finding a memory reference's target in the cache (the hit ratio).
2. Minimizing the time to access information that is residing in the cache (access time).
3. Minimizing the delay time due to a miss, and
4. Minimizing the overheads of updating main memory, maintaining multi-cache consistency, etc.

2.3.1 Locality in Dataflow Programs

Cache has proven its effectiveness when localities govern the structure of the underlying programs. Therefore, it will be beneficial to investigate the presence of localities in dataflow programs and analyze whether cache can be incorporated into dataflow architectures.

If we consider the body of a loop containing a locality pattern, then the complete execution of the loop appears as a number of repetitions of that pattern. These repetitions may be partially distinct (e.g., DOACROSS) or they may overlap (e.g., DOALL) depending on the data dependencies of the loop and the underlying dataflow architecture. In a sequential environment, the instructions of a loop are reused in successive iterations. If instructions are similarly reused in a dataflow environment, temporal locality will exist.

Straight-line code may also produce spatial locality in a dataflow environment. In fact, any section of the code may produce several exploitable spatial localities. A set of instructions which would constitute a spatial locality is grouped together in the virtual address space, and is represented by a path of activity determined by the data dependencies in the program — i.e., a dataflow graph is partitioned into a collection of vertical layers, where nodes in each layer are serially dependent on each other.
2.3.2. Limits of Dataflow Multiprocessing

It was argued in [27] that one of the limitations of dataflow multiprocessing is the amount of latency that can be tolerated. As discussed in chapter one, the success of multithreaded dataflow depends on how quickly context switching can be supported. This is only possible if threads are resident in fast memories, such as cache. The sizes of cache are usually small, hence the number of active threads and thus the amount of latency that can be tolerated is limited. Simulation results from [89] have shown that the best performance is obtained when the number of active contexts (threads) is equal to the maximum number of contexts that can be accommodated by the cache. Increasing the amount of active contexts beyond this limit degrades the performance.

Two possible methods can be utilized to address this limitation. The first method requires some kind of process control that limits the number of active processes, or a scheduling scheme that prioritizes threads that have their data and instructions resident in cache. The other method requires prefetch and replacement policies that ensure that threads that are enabled have their data and instructions already in cache. TAM utilizes the first method, by using a storage directed scheduling scheme [25]. TAM defines a scheduling hierarchy that is closely related to the storage hierarchy. Threads of a function activation are scheduled dynamically by primitive control operations. A TAM program is a collection of code-blocks, roughly corresponding to functions in the source text. Each code-block in turn consists of a number of threads. When a code-block is invoked, an activation frame is allocated to provide local storage. TAM biases the scheduling toward logically related threads: when a thread completes another enabled thread of the same activation is scheduled if possible. When no thread is enabled for the current activation, another activation frame is scheduled. A collection of threads of an activation that execute together is called a quantum. This scheduling policy increases the probability that frame accesses will be in the cache.

The Multi-Threaded Architecture (MTA) proposes using a Register-Use Cache (RU-cache) which keeps track of which register set is assigned to which function instance [41]. This applies to microprocessors such as SPARC and AMD29050, which have large register files that can be partitioned into multiple register sets such that each set services one function activation. A register file with n register
sets will have an RU-cache of $n$ entries. Each entry corresponds to a register set and contains the function pointer (FP) of the function instance to which a register set is assigned. Once a thread is enabled, the RU-cache is associatively searched for a frame pointer which matches the frame pointer value of the ready thread. A match indicates that the thread should be prioritized. Hence, there is a high probability that once a thread is executed, its data will be resident in a register set.

The second method utilized to address this limitation, can be implemented in two possible ways. First, a prefetching policy could be used. A working set, similar to cache in conventional control-flow computers, can be utilized in order to prefetch the other blocks that have a high probability of being referenced in the future [53]. Once a block in the working set is referenced, the whole working set is brought into the cache. Hence, future references to blocks within the working set can be satisfied through the cache, thereby minimizing cache misses. In fact, it was shown that the reference streams of programs executed in TAM could be characterized by a working set function which is similar to those associated with uniprocessor, single threaded programs [66]. Another prefetching scheme would be to delay the entry of an enabled thread into the thread queue until its data and instructions are in the cache — if the data and instructions are found to be in the cache, then the thread would be entered into the thread queue immediately [31]. This would ensure that all enabled threads in the thread queue have their needed blocks in the cache.

Second, a replacement policy could be used. For the instruction cache, additional space within each cache block would be used to hold information, such as a process count, which is the number of activation frames that refer to a particular block [53]. A cache block that is used by a large number of activation frames is a poor candidate for replacement. This would be effective for loop iterations, which contain temporal locality — code needed for each iteration is reused. Additional information that could be predicted at compile time can also be used to determine if a block has a high probability of being used in the near future.

The limitation of dataflow multiprocessing presents the importance of developing a thread scheduling scheme together with appropriate prefetching and replacement policies in order to utilize the resources of a processor, such as cache, efficiently and hence, improve its performance. A combination of
these methods could then be used to optimize the efficiency of cache in a multithreaded dataflow environment.

2.3.3. Cache Memory Designs with ETS

The design of operand and instruction caches in the Explicit Token Store (ETS) dataflow model have been explored in [53]. In ETS, a program is comprised of a collection of code-blocks (disjoint subgraphs) where each of the code-blocks could represent a loop iteration or a function. When a code-block is invoked, a block of memory, called an activation frame, is allocated to store and match the operands that are destined for the instructions in the code-block. In principle, there can be several activation frames associated with a code-block, representing the simultaneous execution of loop iterations. As with other dynamic dataflow models, ETS tokens carry a tag, consisting of an instruction pointer (IP) and a frame pointer (FP). The IP is an offset into the code-block identifying the instruction for which the data (token) is destined. The FP is the base address of the frame identifying the target iteration of the code-block. Each instruction (identified by the IP) contains an offset \( r \) within an activation frame where the match will take place, and one or more displacements that define the destination instructions that will receive the result token(s). Hence, \( FP+r \) is the memory location where the tokens for the instruction are matched [72]. Each destination is also accompanied by an input port (left/right) indicator that specifies the appropriate input arc for a destination actor. This is referred to as a direct matching scheme.

To illustrate the operations of direct matching in more detail, consider the token matching scheme used in Monsoon [72]. Direct matching of tokens in Monsoon is based on the Explicit Token Store (ETS) model. An example of the ETS code-block invocation and its corresponding Instruction and Frame Memory is shown in Figure 2.8. When a token arrives at an actor (e.g., ADD), the IP part of the continuation points to the instruction that contains an offset \( r \) as well as displacement(s) for the destination instruction(s). The actual matching process is achieved by checking the disposition of the slot in the Frame Memory pointed to by \( FP+r \). If the slot is empty, the value of the token is written in the slot and its presence bit is set to indicate that the slot is full. If the slot is already full, the value is extracted, leaving the slot empty, and the corresponding instruction is executed. The result token(s) generated from the
operation is communicated to the destination instruction(s) by updating the IP according to the displacement(s) encoded in the instruction (e.g., execution of the ADD operation produces two result tokens <FP.IP+1, 3.55> and <FP.IP+2, 3.55>).

2.3.3.1. Instruction Cache in ETS

Figure 2.9 shows the detailed structure of the instruction cache. The structure is very similar to a conventional set associative cache, except for the additional information maintained. The low order bits of the instruction address (IP) are used to map instruction blocks into $N$ sets, within each set, search for a block is done associatively using the higher order bits. Each block in the cache contains the following information:

- **Tag**: It represents a block address which identifies a main memory address.
- **Valid-bit**: This bit serves the same purpose as in conventional caches. Initially it is set to zero before the block is allocated. It is changed to one within the block when it is loaded.
- **Process Count**: Represents the number of activation frames that refers to a block. This information is used in instruction cache replacement: an instruction block with the lowest process count is the candidate for replacement.

ETS instructions within a code-block can be reordered to increase locality. In the study reported in [53], the instructions are reordered based on the time of availability of operands. This was done by grouping instructions into execution levels or E-levels [93]. Instructions that become ready (i.e., all inputs
Figure 2.8. ETS representation of a dataflow program execution.
Figure 2.9. Instruction Cache Organization.
are available) at the same time unit are said to be in the same level. Instructions at level 0 for example, are ready for execution at time unit zero. Similarly, those at level 1 become ready for execution at time unit one and so on. E-level ordering is one way of achieving instruction locality.

The instruction memory is then partitioned into blocks and working sets. Blocking is defined to achieve compatibility with the memory bandwidth. Working set defines the average number of instructions that are data independent (and hence can be executed without interruptions). Thus the instructions in a working set comprise the set of instructions that should be prefetched. Block size and working set size are optimized for a given cache implementation to achieve a desired performance. While the optimum working set depends on the program, it was found that working sets of 4 to 8 instructions yield significant performance improvements [53].

It was observed that the instruction cache for the ETS architecture behaves in a manner similar to conventional instruction cache memories. The similarities include the performance behavior of the cache due to variations in the total cache size, set associativity, and cache block size. The cache replacement strategy was based on the number of processes that used a code block. A code block typically consists of instructions belonging to a loop, which are used by a number of activations. A code block with the largest number of active processes will be an unlikely candidate for replacement. This strategy eliminated between 20\% to 70\% of the cache misses as compared to a random replacement strategy.

### 2.3.3.2. Operand Cache in ETS

The design of an operand cache is more complex in ETS like architectures. A two-level set-associative design for the operand cache was proposed in [53] as shown in Figure 2.10. The two levels of associativity result from the need to maintain the association of operands for instructions within a context, and to maintain the association of multiple invocations of the same context. At the first level of associativity, the operand cache is organized as a set of superblocks. Each active context (activation frame associated with a code-block) occupies a superblock. The second level of associativity is used for accessing individual locations within a frame. A superblock consists of the following information:
Figure 2.10. Operand Cache Organization.
- **Cold bit**: Used to indicate if the superblock is occupied or not. This information is used to eliminate misses due to cold starts. In the dataflow model, since the first operand to arrive, will be stored (written), there is no need to fetch an empty location from memory. The cold bit with a superblock is used to allocate an entire frame (or context), and set when the first operand is written to the frame.

- **Tag**: Serves to identify the context (or frame) that occupies the superblock. This is based on the FP address obtained from a token tag.

- **Working set identifiers**: The memory locations within an activation frame (used for token matching) are divided into blocks and working sets, paralleling the blocks and working sets of the instructions in the code block. Thus, a superblock contains more than one working set, and these are accessed associatively (the second level of set associativity). Each working set of a superblock also contains a cold start bit. This bit is used to eliminate unnecessary fetches from memory when the operands are being stored in the activation frame.

A few replacement algorithms for replacing working sets within a superblock and for replacing superblocks themselves have been explored. For working set replacement, a *used words* policy was used. This policy replaces working sets containing memory locations already used for matching operands (hence will not be used in this activation). For superblock replacement, the *dead context replacement* policy that replaces a superblock representing a completed context (or frame) was used.

The operand cache must accommodate several contexts corresponding to different loop iterations, as well as contexts belonging to other code-blocks. In order to minimize the possibility of thrashing, the number of active contexts must be carefully managed (process control). The number of active contexts will depend on the cache size and the size of an activation frame. By reusing locations within a frame, the size of an activation frame can be reduced as well as increasing the process count [52].

The effect on cache miss ratio was explored by varying the number of active processes. It was observed that for an operand cache with $k$-way associativity and $N$ sets, the optimal number of processes is $N^*k$. It was also observed that the use of the dead context replacement policy for replacement of superblocks produced as much as 70% improvement over random replacement strategies. In addition, the use of used words replacement policy in replacing working sets within a superblock produced between 3%...
and 9% improvements over random replacement strategies. Finally the use of cold start bits with operand locations that are yet to be defined showed improvements between 40% to 100% in eliminating cold start cache misses.

2.4. Summary

In this chapter, the concept of dataflow was discussed. The evolution of dataflow architectures from a pure dataflow model with specific hardware support, to a multithreaded dataflow model which incorporates some control-flow methods into the dataflow approach, to alleviate some of the inefficiencies with the pure dataflow method, has also been addressed. One trend in implementing this new model is the utilization of conventional RISC processors with appropriate compilation and program representation schemes in order to support the dataflow execution scheme. This has lead researchers to investigate how control-flow issues can be incorporated into the dataflow approach. One particular area of interest is the incorporation of cache in a dataflow environment.

Since localities within a program exploit the effectiveness of cache, we have shown that localities can exist in dataflow programs as well. Hence, cache can potentially be effective in a dataflow environment. However, before cache can be effectively used in a dataflow environment, several issues must still be resolved. First, the detection and enhancement of localities in a dataflow program is essential. A scheme that detects localities in the dataflow graphs, partitions a dataflow graph into subgraphs while preserving the localities and distribute subgraphs among the processors in a multiprocessor dataflow environment is needed.

Second, the limited amount of storage in a processor (cache) can severely limit the amount of latency that can be tolerated. This could be alleviated by introducing thread scheduling schemes (or process control) that would give priority to threads that already have their information in cache. The storage directed scheduling scheme implemented in TAM tries to execute all the active threads in the current activation first before switching to another activation. This increases the probability that the data and instructions needed by the threads are in the cache. However, this alone is not enough to ensure the effective use of cache. This leads us to the third and last issue, which is appropriate fetching/prefetching.
and placement/replacement policies for caches in a dataflow environment. These policies have to be tailored to the dataflow execution model in order for them to be effective. Use of statistical replacement algorithms for conventional caches, such as least-recently used (LRU) are apt to cause misplacement, if used for instruction and operand memories in a dataflow environment.

An implementation example of an ETS dataflow model with cache memories was discussed. The results from this work have shown the feasibility of the application of cache within the scope of a dataflow environment. It was shown that proper prefetching and replacement policies tailored for the dataflow execution model can effectively increase the performance of cache. The potential performance improvement brought about by incorporation of cache in a dataflow architecture was also addressed. This leads us to believe that the proper implementation of cache in a dataflow environment would produce a viable candidate for a high performance CPU. However, before this can be attained, issues such as appropriate fetching/prefetching and placement/replacement policies have to be investigated.

In the next chapter, we will show how thread scheduling alleviates the limitation of dataflow multiprocessing. In addition, the issue of program partitioning and allocation as well as the schemes that have been advanced in current dataflow projects will be discussed. Finally, we will also see how program partitioning detects and enhances localities in a dataflow program.
3.1. Overview

In the previous chapter, approaches that addressed the limitations of dataflow multiprocessing were presented. We saw that the limited amount of storage in a processor (cache) can severely limit the amount of latency that can be tolerated. These approaches are only directed towards how cache should be implemented in a dataflow environment. Another means of addressing this limitation is to enhance the locality in a dataflow program by reordering nodes or instructions and partitioning programs into threads.

In this chapter, we present methods that perform such enhancements. We first stress the importance of partitioning programs into threads as a means of tolerating high latency operations, maximizing parallelism and minimizing overhead. Next, the allocation problem and the importance of developing thread allocation schemes to optimize the performance of programs in multiprocessors is discussed. Examples of how current dataflow projects perform such an allocation are examined as well. Furthermore, an effective allocation scheme, namely the Vertically Layered Allocation Scheme, which performs both thread partitioning and allocation, is discussed. We will show how this scheme effectively detects spatial localities in a dataflow graph. We then address the importance of loop scheduling and allocation and present a survey of loop scheduling schemes that further improve the performance of executing programs in multiprocessors. An effective DOACROSS loop allocation policy, the Staggered Distribution Scheme, as a part of our previous research work is discussed. Finally, a comparative analysis of all loop scheduling schemes is presented and the distinguishing features of each scheme is discussed.
3.2. Partitioning Programs into Threads

An important issue is the partitioning of programs into multiple sequential threads. Partitioning programs into threads is important since a thread defines the granularity of a computation and thus the basic unit of work for scheduling. Due to the fact that each thread has an associated cost, it directly affects the amount of overhead required for synchronization and context switching — parameters needed by the cache manager to enforce the prefetching and replacement policies. Therefore, any partitioning algorithm should attempt to maximize parallelism while minimizing the overhead required to support the threads.

There have been a number of proposals based on the control-flow model that use multithreading as means of tolerating high latency operations. For example, in multiple contexts schemes used in [101], threads are obtained from subdividing a parallel loop into a number of sequential processes, and context switching occurs when a main memory access is required (due to a cache miss). As a consequence, the granularity of the threads tends to be coarse, thereby limiting the amount of parallelism that can be exposed. On the other hand, non-strict functional languages for dataflow architectures, such as Id, complicate partitioning due to feedback dependencies that may only be resolved dynamically. These situations arise due to the fact that functions or arbitrary expressions can possibly return results before all operands are computed. Therefore, a more restrictive constraint should be placed on partitioning programs written in non-strict languages into threads.

In partitioning a program, several issues should be considered:

1. A partitioning method should maximize the exploitable parallelism. In other words, the attempt to aggregate instructions should not restrict or limit parallelism. Instructions that can be grouped into a thread should be the parts of a code where little or no exploitable parallelism exists.

2. The longer the thread, the longer the interval between context switches. This also increases the locality for better utilization of the processor's resources.

3. Any arc (i.e., data dependency) crossing thread boundaries should be used to improve the spatial locality and/or performance during execution time.
There are a number of partitioning algorithms that convert dataflow graph representations of programs into threads. Traub and Culler [99] have proposed one such partitioning strategy to address these issues. They define a thread as a subset of the instructions comprising a procedure body, such that:

1. A compile-time instruction ordering can be determined for the thread which is valid for all contexts in which the containing procedure can be invoked, and
2. Once the first instruction in a thread is executed, it is always possible to execute each of the remaining instructions, in the compile-time ordering, without pause, interruption, or execution of instructions from other threads.

They have proposed partitioning and global analysis algorithms that meet these two requirements. Schauser et al. [87] proposed a similar strategy by compiling a lenient parallel language, Id90 for conventional processors using compiler-controlled multithreading. Their approach involves several large translation steps: Id90 to dataflow graphs, program graphs to dual graphs, dual graphs to TAM threads, and TAM threads to native machine code. A dual graph is a directed graph with three types of arcs: data, control, and dependence. A data arc represents the data dependency between producer and consumer nodes. A control arc represents the scheduling order between nodes, and a dependence arc specifies long latency operation due to message handlers sending/receiving messages across code-block boundaries. Dual graphs facilitate the synthesis of control operations and management of storage. They make the flow of control and data explicit and, by retaining both in graphical form, allowing transformations to be applied to one without prematurely constraining the other. The partitioning algorithm first identifies small safe partitions. Then these basic partitions are iteratively merged into larger safe partitions by applying simple merge rules, eliminating redundant control arcs, and combining switches and merges until the process converges. A partition is safe if:

1. No output of the partition needs to be produced before all inputs to the body are available.
2. When the inputs to the body are available, all nodes in the body are executed in a sequential fashion, and
3. No arc connects a body node to an input node of the same partition.
Once the program is partitioned into threads, threads should be allocated or distributed among the processing elements in order to exploit the maximum parallelism in the program. The allocation problem and how allocation is performed in current dataflow projects will be discussed in the next section.

3.3. Allocation Problem

Similar to the control-flow multiprocessors, the issue of task partitioning and allocation is also of major interest to multithreaded dataflow architectures. The goal is to exploit the maximum concurrency of a program graph by minimizing contention and communication for processing resources. The literature has addressed two general directions to the issue of program allocation, namely static and dynamic schemes [60]. In a static scheme, the tasks are allocated at compile-time using global information about the program behavior and the system organization. The cost of allocating tasks is incurred once for a given program even though the program may be executed repeatedly. However, static allocation policies can be inefficient when estimates of run-time dependent characteristics are inaccurate. A dynamic allocation policy on the other hand is based on measuring processor loads at run-time, assigning activated tasks to the least loaded processor and balancing the load by migrating the tasks. The disadvantage of dynamic allocation is the overhead involved in determining processor loads and allocation of tasks at run-time. It has been shown that obtaining an optimal allocation of a graph with precedences is NP-complete [76]. Therefore, heuristic solutions are the only possible approach to solving the allocation problem suboptimally in polynomial time.

3.3.1. Program Allocation on Dataflow Machines

Architectures such as Monsoon and EM-4 have adopted a dynamic load balancing scheme to distribute work among processing elements. In Monsoon, work distribution is performed by the frame manager, which also handles the management of activation frame memory [39]. When a procedure calls another procedure, the frame manager chooses the processor on which to allocate the frame, so that the distribution of the workload does not have to be specified in the compiled code. The frame manager
distributes work based on a code-block granularity in a round-robin fashion — a code-block roughly corresponds to a procedure or a loop iteration. Each processor has a set of round-robin counters, one for each frame size that it uses to distribute work to other processors. Each count on the counters corresponds to a specific processor. Since each processor has its own round-robin counters, work distribution decisions are made locally by looking up the current value on the counter for a specific frame size. This value would determine which processor to assign the new frame. The counter is then incremented. Loops must be specified as either sequential, bounded, or unbounded. The compiler generates specific code for each kind of loop. A sequential loop executes one iteration at a time and uses only one frame. A bounded loop executes a specified number \(k\) of iterations in parallel, and uses \(k\) frames. An unbounded loop is compiled into a recursive call so all loop iterations can potentially execute in parallel, even if there is data dependency among iterations. Hence, the programmer is in charge of specifying how the loop is to be executed. Since each iteration of a bounded or unbounded loop uses one frame, the same frame management and distribution scheme, discussed earlier, is used to distribute loop iterations among processors. Results on the Monsoon hardware have shown that this scheme balances the load reasonably well [39].

Although this scheme balances the load, it does not take into account the effect of communication cost in sending the arguments or result of one code-block to another. If the communication cost of sending a result from one code-block to another is high such that executing both code-blocks in the same processor results in faster execution, then any benefits attained by balancing the load is lost. This is critical for bounded and unbounded loops. If there is a dependence between iterations, then the amount of communication cost that is incurred is staggering, since each iteration of a loop is potentially allocated to a different processor. Results have shown that the processors' idle cycles tend to be the largest contributor to the increased total cycle counts on Monsoon. One possible cause could be the lack of work on one or more processors, due to the lack of parallelism: i) during startup and termination of a program, or ii) in the algorithm.

In the EM-4, arcs in the dataflow graph are divided into normal arcs and strongly connected arcs [102]. Dataflow subgraphs whose nodes are connected by strongly connected arcs are called strongly connected blocks. The execution strategy for strongly connected blocks is that once the execution of a
block begins, the PE would exclusively carry out the execution of the nodes of the block until its completion. Therefore, a strongly connected block acts as a macro node which includes several instructions and is executed as if it was a single node. A strongly connected block could correspond roughly to a code-block or a sequential thread, where sequential execution of the block is more efficient than parallel execution.

PEs, in the EM-4, are grouped together and each group is connected by a local connection path which does not intersect with the local connection path of any other group. PEs allocated to different groups communicate via another path called the inter-group path.

The EM-4 can handle load distribution on three levels: instruction level, block level, and function level [54]. The instruction level activity is represented as a dataflow graph and the block level activity is represented as a strongly connected block. Since the structure of the dataflow graph and strongly connected blocks are compile time events, the instruction and block level partitioning is determined statically. At the function level, both static and dynamic distributions have been enforced. For static distribution, the best result was obtained from a method called relative PE number method [54]. This method allocates one callee function to the next PE in the group and the other to the next PE out of the group. The dynamic mechanism on the other hand, detects the least loaded PE within a group and assigns the execution of a function to that PE. The preliminary performance analysis has shown that the static distribution policy does not work well when the topology of the program fails to match the topology of the hardware structure [54]. Hence, behavior of the program should be known in advance in order to determine the proper distribution scheme to use. For dynamic function distribution naturally, the load on the PEs are more equally distributed than with the static scheme. However, performance results on the EM-4 have shown that this does not necessarily result in a better execution time.

Similar to Monsoon, the EM-4 distributes functions and loop iterations with no concern for the overhead due to the communication cost. Although, communication locality is used as a means to allocate a function, it does not take into consideration that the cost of executing a function/iteration in a different PE could increase the communication overhead and hence degrade the overall performance. Both Monsoon and EM-4 utilize a dynamic allocation/distribution scheme which increases the execution overhead, and does not take into consideration the execution behavior of a program in determining the
allocation. An allocation policy that analyzes the program execution pattern in making its allocation with respect to the underlying hardware characteristics, such as number of PEs and communication cost, would generally result in better performance.

3.3.2. Vertically Layered Allocation Scheme

The Vertically Layered (VL) allocation scheme [61] was developed based on a compromise between computation and communication costs. It performs both thread partitioning and allocation. The input to VL is a program graph, which is a DAG representation of a program \( G = G(N, A) \), where \( N \) represents the set of instructions and \( A \) represents the partial ordering \(<\ast\>\) between the instructions. A directed path from node \( n_i \) to node \( n_j \) implies that \( n_i \) precedes \( n_j \) (i.e., \( n_i <\ast n_j \)). In addition, an expected execution time \( t_i \) is associated with every node \( n_i \in N \) and a communication cost \( c_{ij} \) is considered for every arc \( a(n_i, n_j) \in A \). An example of a program graph is shown in Figure 3.1 with the execution time \( t_i \) in bold letters.

The VL allocation scheme consists of two separate phases: The \textit{separation} phase and the \textit{optimization} phase. In the separation phase, a program graph is first partitioned into disjoint horizontal layers \( H \) such that the nodes in each layer \( H_i (i = 1, 2, \ldots, k) \) can be performed in parallel and the layers are linearly ordered with respect to their precedence constraints (i.e., \( H_i <\ast H_{i+1}, 1 \leq i \leq k-1 \)). This is similar to the E-levels that was discussed in chapter 2. The resulting program graph is then partitioned into vertical layers based only on the execution times \( T \), where each vertical layer consists of one or more serially connected set of nodes (thread) which is considered for assignment to a processing element. To determine the appropriate vertical layers, the critical path of a directed graph \( G \) is first identified. The critical path defines all the critical nodes; therefore, by assigning the nodes that lie on the critical path to a
Figure 3.1. An example of a dataflow graph.
single vertical layer, the communication overhead associated with critical nodes is minimized. However, special provisions must be taken to handle conditional nodes and loops before the critical path of an arbitrary directed graph $G$ can be determined. To facilitate the handling of conditional nodes and loops, an approximate method for determining the critical path is utilized. Without loss of generality, only the expected execution times $T$ are considered in determining the critical path.

Conditional nodes are used to handle two-way selection (i.e., "if-then-else") and contains a predicate and two alternative subgraphs corresponding to true or false branches. Since the result of the predicate is known at run-time, a probability is assigned to each of the alternative subgraphs such that $p_{\text{true}} + p_{\text{false}} = 1$. The execution times of the two alternative subgraphs are then multiplied with their respective probabilities to obtain their expected execution times.

For loop graphs, the execution time of each node in the loop is multiplied with the expected number of iterations of the loop. The loop is then serialized to avoid having some nodes being assigned to a different processor. These expected probabilities and number of iterations may be defined explicitly by the programmer, derived by the compiler, or refined by monitoring previous executions of program graphs.

Once these expected execution times are assigned, the approximate critical path of the program graph is determined. For simplicity, if no unique critical path exists, the algorithm arbitrarily chooses one. These nodes will be defined as belonging to the vertical layer $V_{\text{CP}}$. At the same time the set of nodes $V_{\text{CP}}$ is queued in a First In First Out (FIFO) queue $Q$ according to their precedence relationship. For program graphs with vertical layers $V_m$ (for $m = 1, 2, ..., \max$), the set of nodes $V_{\text{CP}}$ is assigned to the vertical layer $m = \left\lfloor \frac{\max}{2} \right\rfloor$. All other nodes $n_i \notin V_{\text{CP}}$ are also rearranged in an iterative manner as follows: Let $V_{s-1}$ represent the set of nodes which have already been rearranged into vertical layers at the $(s-1)$th step (initially we have $V_0 = V_{\text{CP}}$). Next, remove a node $n_i$ from the queue $Q$ and then find a set of nodes $V_{\text{LDP}}^S$ that forms the longest directed path for every output arc emanating from $n_i$ such that

$$V_{\text{LDP}}^S \cap V_{s-1} = \emptyset \quad \text{and} \quad V_{\text{LDP}}^S \cup V_{s-1} = V^S.$$  

Note that finding the longest directed path emanating from an arc in node $n_i$ involves the same procedure as determining the critical path with the nodes in the set $V_{s-1}$ removed. To implement this, each node in
V\textsuperscript{LDP}\textsubscript{S}, as it is inserted into the queue, is marked to indicate that it has already been considered for allocation.

Each set of nodes V\textsuperscript{LDP}\textsubscript{S} is then assigned to the first available vertical layer. The procedure used to determine whether a vertical layer is available for allocation is based on how densely the nodes can be assigned to a vertical layer. For a given number of processors \( p_{\text{max}} \), a density factor \( D \) keeps track of the largest number of nodes assigned to a particular horizontal/vertical intersection. When a set of nodes is considered for allocation to a particular vertical layer, the number of nodes already assigned to each horizontal/vertical intersection must be less than the density factor \( D \) for all \( n_i \in V\textsuperscript{LDP}\textsubscript{S} \). If the search process fails for all the vertical layers, the density factor \( D \) is incremented by 1 and the process is repeated. This ensures that all of the layers will be arranged as densely as possible. The separation phase is completed when the queue is empty and all the nodes in \( G \) are rearranged into vertical layers. An example of a vertically layered graph of Figure 3.1 is shown in Figure 3.2.

In the optimization phase, the Communication to execution Time Ratio (CTR) heuristic is used to further optimize the allocation by considering the inter-PE communication costs. This is done by considering whether the inter-PE communication overhead offsets the advantage gained by overlapping the execution of two subsets of nodes in separate processing elements. This process is repeated in an iterative manner until no improvement in performance can be obtained by combining two subsets of nodes allocated to different processors.

The CTR heuristic considers two types of inter-PE communication behavior for optimization — Type A and Type B. Type A is associated with the transitory relationship between two subsets of nodes in \( G \) that are grouped in two distinct vertical layers. For example, consider two subsets of nodes \( N_\alpha \) and \( N_\beta \) which are assigned to vertical layers \( V_\alpha \) and \( V_\beta \), respectively. If \((n_i \in V_\alpha) < \bullet (n_j \in V_\beta) \) and \((n_l \in V_\beta) < \bullet (n_k \in V_\alpha)\), then there will be inter-PE communication costs associated with the execution of the two
Figure 3.2. Vertically layered graph of Figure 3.1.
vertical layers which may delay the execution time of the vertical layer \( V_\alpha \). Type B, on the other hand, is a more general case of Type A behavior.

Consider an example of Type A inter-PE communication behavior as shown in Figure 3.3. Assume \( T_\alpha \) and \( T_\beta \) are the total execution costs associated with \( N_\alpha \) and \( N_\beta \), respectively, where

\[
T_\alpha = \sum_{i \in N_\alpha} t_i \quad \text{and} \quad T_\beta = \sum_{i \in N_\beta} t_i
\]

If there is a transitory relation between \( N_\alpha \) and \( N_\beta \), we can associate with it a communication cost \( C_{\alpha\beta} \), where

\[
C_{\alpha\beta} = c_{\alpha\beta} + c_{\beta\alpha} = 2c_{\alpha\beta}
\]

With these parameters three cases are possible:

Case 1: \( T_\alpha > T_\beta + C_{\alpha\beta} \). Therefore, the initial assignment of the vertical layers to two distinct PEs will not affect the overall execution time.

Case 2: \( T_\alpha < T_\beta + C_{\alpha\beta} < T_\beta + T_\alpha \). \( T_\beta \) and the communication cost \( C_{\alpha\beta} \) are significant enough to affect the execution time \( T_\alpha \) and, therefore, the overall execution time. This will result in a new critical path through the subset of nodes \( N_\beta \). In order to improve the overall execution time, we can consider combining the subset of nodes \( N_\alpha \) and \( N_\beta \) into a single vertical layer. This eliminates the communication cost \( C_{\alpha\beta} \) and results in an overall execution time of \( T_\beta + T_\alpha \).

However, if the overall execution time \( T_\beta + T_\alpha \) results in a larger delay compared to executing them in two separate vertical layers, the two subsets are assigned to different PEs.

Case 3: \( T_\beta + T_\alpha < T_\beta + C_{\alpha\beta} \). The execution of \( N_\alpha \) and \( N_\beta \) in a single PE results in superior performance. Therefore, if the ratio of communication to execution cost, \( C_{\alpha\beta} / (T_\beta + T_\alpha) \), is
Figure 3.3. Type A inter-PE communication behavior.
greater than \(1 - T_\beta(T_\beta + T_\alpha)\), an improved total execution time can be obtained by combining \(N_\alpha\) and \(N_\beta\) and executing them on a single PE.

Consider an example of Type B behavior depicted in Figure 3.4. One of the major causes of delay for the new critical path is the relative assignment of node \(n_\beta\). As can be seen, the assignment of \(n_\beta\) to the vertical layer \(V_\beta\) has increased the total execution time. To reduce the inter-PE communication costs, we can consider combining nodes \(n_\beta\) and \(n_\alpha\) into a single vertical layer \(V_\alpha\). Then we have either \(c_{\alpha\beta} < t_\alpha\) or \(t_\beta < c_{\alpha\beta}\). If the communication to execution time ratio, \(c_{\alpha\beta}/t_\alpha\) is greater than unity by combining the nodes into the vertical layer \(V_\alpha\) the result is an incremental improvement of \(c_{\alpha\beta} - t_\beta\) in total execution time.

To apply the CTR heuristic algorithm, we consider the execution time \(T_{CP}\) of a new critical path which includes the effects of inter-PE communication costs. Hence, \(T_{CP}\) corresponds to the total execution time \(T_{EXE}\) after the program graph \(G\) has been vertically layered in the separation phase. We then consider the type of communication behavior, which resulted in \(T_{CP}\). If an improvement in \(T_{CP}\) results after applying the CTR heuristic, the nodes are combined into a single PE. Since combining two parallel subsets of nodes into a single processing element forces them to be executed sequentially, a new critical path may emerge from the optimization process. Therefore, this process is repeated, in an iterative manner, until no improvement in performance can be obtained by combining two subsets of nodes associated with \(T_{CP}\).

Performance analysis indicates that this scheme is very effective in reducing the communications overhead. On average, the VL allocation scheme showed promising improvement over the Critical Path list schedule in the presence of inter-PE communication delays when a sufficient number of PEs are available [61]. Nevertheless, the VL allocation scheme does not take into consideration the large amount of parallelism present in loops. As a result, loop graphs are not unfolded according to the number of iterations, instead the loop graph is considered as a node with an execution time proportion to the number of iterations. This will force all iterations to be executed in one processing element — serial execution of
Figure 3.4. Type B inter-PE communication behavior.
independent iterations. A more elegant approach should allow loop unfolding and distribution of loop iterations among the processing elements with respect to the communication cost.

An analysis of static dataflow graph representation of a program reveals only a portion of the potential parallelism. The fact that dynamic parallelism contains the largest source of parallelism in a program is reason enough to properly address it. An allocation scheme capable of detecting and allocating dynamic parallelism during compile-time — expansion and distribution of loops — would effectively reduce run-time overhead, maximize parallelism, and would lead to improved performance. In order to address this limitation, an extension to the VL allocation scheme (loop allocation) is needed, which will be discussed in the next section.

3.4. Loop Allocation

Since loops are the largest source of parallelism [78] considerable attention has been paid to the partitioning and allocation of loop iterations among processors in a multiprocessor environment. The key goal is to maximize parallelism while minimizing the processor load imbalances and network communication.

The literature abounds with scheduling algorithms for loops. These algorithms can be categorized as static and dynamic [55]. In static scheduling, the division of iterations among the processors is determined prior to the execution time. This results in a low run-time scheduling overhead. On the other hand, static scheduling can cause unbalanced distribution of load among the processors if the execution times of individual iterations vary. The variance in execution can result from conditional statements [42], or because of interference from the environment (the operating system, switching between iterations or time-sharing with other programs). Dynamic scheduling determines the division of iterations among processors at run-time. Some algorithms may dynamically reassign iterations to different processors based on the progress made by processors on previously assigned iterations. Thus, dynamic schemes can achieve better load balance, but this comes at the expense of run-time scheduling overhead.

Loops can be categorized as sequential loops, vector loops (DOALL), and loops of intermediate parallelism (DOACROSS) [28]. For a DOALL loop, all $N$ iterations of the loop can be executed
simultaneously. When there is a sufficient number of processors, all iterations can be executed in parallel. But with a finite number of processors, iterations must be divided among the processors. When iterations of a loop must be executed completely sequentially (Sequential loops), no improvement can be gained by using multiple processors. However, some loops may exhibit intermediate levels of parallelism permitting some overlapped execution among iterations.

The DOACROSS loop model proposed by Cytron [28] can mimic sequential loops, vector loops or loops with intermediate levels of parallelism. Iterations may be either data or control dependent on other iterations. Control dependencies are caused by conditional statements. Data dependence appears in the form of sharing results computed by other iterations. Data dependence can be either lexically-forward (data from higher indices used by iterations with lower indices) or lexically-backward (data from lower indices used by iteration with higher indices). Normally, lexically forward dependencies (LFD) do not contribute to delays in executing loop iterations. Sometimes a lexically-backward dependence (LBD) can be transformed into a lexically-forward dependence by reordering the statements of the loop, provided the statements do not form a dependence cycle [28]. DOACROSS loops where the LBD cannot be transformed into LFD lead to delays in executing successive iterations. Such loops are the subject of most research.

3.4.1. Loop Scheduling Algorithms For DOALL Loops

Static scheduling schemes assign a fixed number of loop iterations to each processor. For a loop with $N$ iterations executed on $P$ processors, each processor will receive a total of $\lceil N/P \rceil$ iterations. Variations on how these iterations are distributed among the available processors lead to different algorithms. Block scheduling or static chunking (SC) assigns iterations 1 through $\lceil N/P \rceil$ to the first processor, iteration $\lceil N/P \rceil + 1$ through $2 \times \lceil N/P \rceil$ to the second processor, and so on. Cyclic scheduling allocates iterations $i, i + P, i + 2P, \ldots$, to processor $i$ ($1 \leq i \leq P$).

When the execution times of individual iterations vary, static chunking leads to unbalanced loads among processors. For example, when the execution times of iterations monotonically decrease (i.e., triangular iteration space), the chunks containing smaller iteration indices consume more time than chunks
containing iterations of higher indices. In such a case, the execution time of the DOALL loop is bounded by the completion times of the earlier chunks. Thus static chunking could perform suboptimally, and cause under-utilization of processor resources [42]. Since cyclic scheduling assigns consecutive iterations to different processors, a better load balance across processors is achieved.

The main advantage of static scheduling methods is their simplicity (hence small scheduling overhead). No run-time overhead is incurred by such methods since all scheduling decisions are made at compile time. This implies the availability of information on loop bounds and number of processors. When such information is not known statically (or changes dynamically), static scheduling methods lead to unbalanced workload among processors.

Dynamic scheduling schemes are proposed to address the limitations of static methods. Typically, shared variables and critical sections are used to control the distribution of iterations to idle processors. Thus, an idle processor locks the shared variable and obtains an iteration (or a set of iterations). This leads to run-time overhead, both in terms of the time required to access the shared variable (including communication cost and synchronization cost), and the time needed to compute the next schedule. Complex dynamic schemes with high scheduling costs and large communications costs may negate any performance gained. In order to simplify the analysis, we will assume that the scheduling cost per scheduling step, for all dynamic schemes will be the same, and is given by $T_{sched} = 2C + tsched$, where $C$ is the communication cost for accessing a shared variable as well as the communication cost for returning an updated value to the shared variable, and $tsched$ is the time required to calculate the chunk size. Some of the dynamic scheduling algorithms are discussed below.

3.4.1.1. Self-scheduling

Self-scheduling (SS) [95] is a dynamic scheme that schedules iterations of a loop, one at a time. An idle processor obtains a new iteration and executes it. Hence, processors finish at nearly the same time
and the workload is balanced. However, since this method requires $N$ scheduling steps (one for each iteration), the overall scheduling cost may be unacceptable. In addition, processors may have to contend with synchronization delays in accessing shared variables. For example, with $P$ processors attempting to obtain a loop iteration, one processor must wait $(P - 1)T_{sched}$ waiting for all the other processors to access and update the shared variable. The average wait time for $P$ processors is given by $P(P - 1)T_{sched}/2$. With $N$ iterations, the average wait-time per processor is given by $N(P - 1)T_{sched}/2$.

3.4.1.2. Fixed-size chunking

In an attempt to reduce the number of scheduling steps needed, \textit{Fixed-size chunking} (FS) schedules a fixed number of iterations to each idle processor (as opposed to one iteration in SS) [57]. This reduces scheduling overhead, but the trade-off is increased load imbalance due to coarser task granularity. It is often difficult to determine the optimal number of iterations to schedule at each step. Small chunks increase the number of scheduling steps (hence scheduling overhead), while large chunks may cause unbalanced load across processors. Kruskal and Weiss [57] have proposed a scheme to calculate an optimal chunk size based on the number of iterations, the number of processors, the standard deviation of the execution times of individual iterations, and the scheduling overhead. Since it is often difficult to determine the variance among the iteration execution times before executing them and because the variance may depend on the environment of the processor to which they are assigned, this method is not practical for real applications.

Several schemes have been proposed to minimize the limitations suffered by both self-scheduling and fixed-size chunking [42, 77, 100]. These schemes are based on scheduling chunks with decreasing number of iterations. Typically, larger chunks are initially scheduled, reducing the scheduling overhead, while smaller chunks are subsequently scheduled to smooth any load imbalances resulting from previous assignments.

3.4.1.3. Guided Self-scheduling
In **Guided self-scheduling** (GSS), the size of the chunk scheduled on the next idle processor is \( \lceil R/P \rceil \), where \( R \) is the number of remaining iterations [77]. Thus, the chunk size varies from \( \lceil N/P \rceil \) iterations down to one iteration. This algorithm allocates large chunks at the beginning of a loop's execution to reduce the scheduling overhead. Smaller chunks are then allocated as the number of remaining iterations to be executed decreases. The last \( P - 1 \) chunks consist of one iteration that can be used to balance the load, thus increasing the likelihood that all processors finish at the same time. A feature of GSS is that approximately two-thirds of the remaining iterations are allocated over every \( P \) chunks [42]. For example, if there are \( N = 100 \) iterations to be executed on a four-processor system, the size of the chunks are: 25, 19, 14, 11, 8, 6, 5, 3, 3, 2, 1, 1, 1, 1. It should be noted that GSS addresses the problem of uneven starting times of processors resulting from the delays in acquiring the chunks. Simulations involving constant-length iterations and uneven processor starting times, as well as iterations with variable-length running times were conducted and found that GSS performs better than SS method [77].

The number of scheduling steps required for GSS, in the best case, is \( P \), when the number of iterations \( N \) is approximately equal to the number of processors \( P \). Otherwise, the maximum number of scheduling steps is \( PH_{\lceil N/P \rceil} \), where \( H_n \approx \ln(n) + \gamma \frac{1}{2n} \) is the \( n^{th} \) harmonic number and the \( \gamma \approx 0.5772157 \) is the Euler's constant [77]. For large \( N \) this approximates to \( P \ln \lceil N/P \rceil \) [103]. The number of scheduling steps required for GSS is more than that for FS, but less than that for SS. Although GSS often achieves a balanced load when iteration execution times vary widely, it is still possible that some initial chunks (due to their large sizes) do not complete by the time all other chunks have completed.

### 3.4.1.4. Factoring

**Factoring** was specifically designed to handle iterations with widely varying execution times [42]. Similar to GSS, this scheduling strategy uses variable and decreasing chunk sizes. At each round, factoring schedules half of the remaining iterations into \( P \) equal sized chunks. In other words, each chunk
contains \( \lceil R/2P \rceil \) iterations, where \( R \) is the number of unscheduled iterations. Factoring allocates smaller initial chunks than GSS, thus, alleviating one of the main problems of GSS. The chunk sizes for \( N = 100 \) iterations to be executed on a four-processor system are: 4 chunks with 13 iterations each, 4 chunks with 6 iterations each, 4 chunks with 3 iterations each, 4 chunks with 2 iterations each, and finally 4 single-iteration chunks. The chunk size for factoring is determined by:

\[
K_j = \left\lceil \frac{1}{2} 2^{j+1} \frac{N}{P} \right\rceil \quad j \leq 0
\]  

(3.1)

where \( K_j \) is the chunk size for factoring step \( j \), \( N \) is the total number of iterations, and \( P \) is the number of processors. The number of scheduling steps can be determined by setting \( K_j \) to one and solving equation (3.1) for \( j \) — the number of factoring steps. However, since factoring schedules \( P \) equal size chunks per batch (factoring step), the total number of scheduling steps is approximately equal to \( P \lceil 1.44 \ln(N/P) \rceil \) [103]. As can be seen, the number of scheduling steps for factoring is 1.44 times that for GSS. However, it has been shown that factoring performs better than GSS when the iteration execution times vary significantly [42].

3.4.1.5. Trapezoid Self-scheduling

*Trapezoid Self-Scheduling* (TSS) is another scheme that is developed for loops with varying iteration execution times [100], by using variable and decreasing chunk sizes. TSS attempts to reduce the synchronization cost of obtaining work by individual processors by simplifying the scheduling computations in the critical section. TSS uses a simple linear function to determine the size of the chunk allocated at each step. This method will rely on a programmable number for the size of the first and final chunks, \( f \) and \( l \). The sizes of the chunks between successive scheduling steps are decreased by \( s = (f - l)/(C - 1) \), where \( C = \lceil 2N/(f + l) \rceil \) is the number of chunks to be scheduled. Thus, the first chunk size is \( c_1 = f \), and the second is \( c_2 = c_1 - s \). In general, \( c_{i+1} = c_i - s \).
The typical values for the first and last chunks are \( f = \left( \frac{N}{2} \right) \) and \( l = 1 \) \([100]\). The number of scheduling steps for trapezoid self-scheduling is equal to the total number of chunks \( C \), which ranges from \( 2P \) to \( 4P \). For large \( N \), the total number of scheduling steps is approximately equal to \( 4P \) \([103]\). TSS allocates smaller initial chunks than GSS, and requires fewer scheduling steps than factoring.

3.4.2. Comparative Analysis Of DOALL Loop Scheduling Schemes

The advantages and disadvantages of various scheduling algorithms are summarized in Table 3.1. As can be seen, fixed-size chunking requires the smallest number of scheduling steps while self-scheduling requires the most. Fixed-chunking is more efficient since the chunk sizes can be determined at compile time. Unlike fixed-chunking, self-scheduling balances the load on processors more evenly, however, the \( N \) scheduling steps needed may offset any performance gains. Since processor must access a shared variable to obtain work, SS also adds delays due to network and memory contention.

Factoring requires more scheduling steps than GSS, but the chunk size is computed less frequently (every \( P \) steps instead of every steps in GSS). Factoring allocates more smaller chunks than GSS in order to balance the load, accounting for the increased number of scheduling steps. The earlier chunks in GSS may take longer to execute than all other chunks, leading to unbalanced load, particularly when the execution time of iterations decreases with increasing indices.

It has been shown that when the ratio of the number of iterations to the number of processors is larger than 55 TSS requires fewer scheduling steps (4\( P \) steps) than that required by GSS, and when the ratio is larger than 16, TSS requires fewer scheduling steps than Factoring \([103]\). This is true, because the next chunk size differs from the current chunk size by a constant, and thus the scheduling computation
### TABLE 3.1. Comparative analysis of DOALL scheduling algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Scheduling steps</th>
<th>Chunk size</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-scheduling (SS)</td>
<td>$N^{(1)}$</td>
<td>1</td>
<td>Can balance the workload well.</td>
<td>Requires $N$ scheduling steps. Should only be used in systems in which the overhead for accessing shared variables is small. Chances of network and memory contention are very high. Contention for network and memory becomes a major problem.</td>
</tr>
<tr>
<td>Fixed-size chunking (FS)</td>
<td>$P^{(2)}$</td>
<td>$\left\lceil \frac{N}{P} \right\rceil$</td>
<td>Requires the minimum number of scheduling steps.</td>
<td>May not balance the workload very well, especially if the variance in iteration execution times is large.</td>
</tr>
<tr>
<td>Guided Self-scheduling (GSS)</td>
<td>$P \left\lceil \ln \frac{N}{P} \right\rceil$</td>
<td>$\left\lceil \frac{R^{(3)}}{P} \right\rceil$</td>
<td>Tradeoff between load balancing and scheduling overhead. Number of scheduling steps between SS and FS, and tries to handle variations in iteration times by balancing the workload.</td>
<td>Early chunk could be so large, it does not complete by the time all other chunks have completed. The current chunk size must be calculated at every step.</td>
</tr>
<tr>
<td>Factoring</td>
<td>$P \left\lfloor 144 \ln \frac{N}{P} \right\rfloor$</td>
<td>$\left\lfloor \frac{R}{2P} \right\rfloor$</td>
<td>Allocates smaller chunks than GSS in order to balance the workload.</td>
<td>Requires more scheduling steps than GSS.</td>
</tr>
<tr>
<td>Trapezoid Self-scheduling (TSS)</td>
<td>$4P$</td>
<td>$c_{p+1} = c_{p} - \frac{s}{(4)}$</td>
<td>The chunk size decreases linearly, hence the difference between the current chunk and the next chunk is constant. The calculation of the current chunk size can be performed in parallel eliminating the need for a critical section. Fewer scheduling steps than GSS and factoring when the iteration-to-processor ratio is larger than 55 and 16 respectively [103].</td>
<td>The algorithm for computing the chunk size is fairly complex. Allocates larger portions of remaining work to later chunks, which may generate large load imbalances for the last few scheduling steps.</td>
</tr>
</tbody>
</table>

(1) Number of iterations
(2) Number of processors
(3) Number of remaining iterations
(4) $s = f - l(C - 1)$, where $C = \left\lceil 2N(f + l) \right\rceil$, and $c_{1} = f$.

The typical values for the first and last chunks are $f = NP$ and $l = 1$ [103].
is simpler. In TSS, even later chunks may remain large, potentially causing load imbalance. GSS and factoring, on the other hand, guarantee that the last $P$ chunks contain only one iteration per chunk. These small chunks can be used to balance the finishing times of all processors.

Performance of GSS, factoring, self-scheduling, and static chunking have been simulated on the RP3 multiprocessor platform for several benchmark loops [42]. This study shows that factoring is scalable, and unlike GSS, its performance is resistant to variance in iteration execution time. In another study, it was shown that GSS did not perform well when the variance in iteration execution times is large (e.g., adjoint convolution programs) [103]. Factoring and TSS balance the workload better than the other methods. These studies also suggest that none of the algorithms perform well when $N$ is small since there is insufficient work to offset the overhead of scheduling. Since the scheduling overhead is minimal for static-chunking and fixed-size chunking, they perform better when the variance among iteration execution times is small. Table 3.2 shows the number of iterations assigned to a processor at each scheduling step for GSS, fixed-size chunking (FS), factoring, and TSS.

### 3.4.3. DOACROSS Loop Scheduling

Chen and Yew [19] have used an event-driven simulator to measure the parallelism inherent in application programs. Six real application programs from the PERFECT Benchmark suite were used in their study. They observed that the loss of parallelism after serializing DOACROSS loops was very significant. This supports the need for suitable schemes for the parallel execution of DOACROSS loops.

DOACROSS loops can be classified as regular and irregular loops. In a regular DOACROSS loop, dependence distances are constant while the dependence distance varies from iteration to iteration in irregular DOACROSS loops. Regular DOACROSS loops are easier to parallelize than irregular loops.
TABLE 3.2. Number of iterations assigned to a processor at each scheduling step with $n = 1000$, $P = 4$.

<table>
<thead>
<tr>
<th>Step</th>
<th>GSS</th>
<th>FS</th>
<th>Factoring</th>
<th>TSS $f = 125$, $l = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>250</td>
<td>250</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>2</td>
<td>188</td>
<td>250</td>
<td>125</td>
<td>117</td>
</tr>
<tr>
<td>3</td>
<td>141</td>
<td>250</td>
<td>125</td>
<td>109</td>
</tr>
<tr>
<td>4</td>
<td>106</td>
<td>250</td>
<td>125</td>
<td>101</td>
</tr>
<tr>
<td>5</td>
<td>79</td>
<td>-</td>
<td>63</td>
<td>93</td>
</tr>
<tr>
<td>6</td>
<td>59</td>
<td>-</td>
<td>63</td>
<td>85</td>
</tr>
<tr>
<td>7</td>
<td>45</td>
<td>-</td>
<td>63</td>
<td>77</td>
</tr>
<tr>
<td>8</td>
<td>33</td>
<td>-</td>
<td>63</td>
<td>69</td>
</tr>
<tr>
<td>9</td>
<td>25</td>
<td>-</td>
<td>31</td>
<td>61</td>
</tr>
<tr>
<td>10</td>
<td>19</td>
<td>-</td>
<td>31</td>
<td>53</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>-</td>
<td>31</td>
<td>45</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>-</td>
<td>31</td>
<td>37</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>-</td>
<td>16</td>
<td>28</td>
</tr>
<tr>
<td>14</td>
<td>6</td>
<td>-</td>
<td>16</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>-</td>
<td>16</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>-</td>
<td>16</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>-</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>-</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>-</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>-</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>-</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>-</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>26</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>27</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>28</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>29</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>31</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
3.4.3.1. The Regular DOACROSS Model

Cytron [28] developed a DOACROSS model for the execution of loops with some degree of parallelism among various iterations. Consider a single loop $L$ with $s$ statements ($S_1, S_2, ..., S_s$) and $N$ iterations. If $T(S_i, S_j)$ is the execution time of statements $S_i$ through $S_j$ ($i < j$), then the DOACROSS model has $d = 0$ for vector loops, $d = T(S_1, S_s)$ for sequential loops, and $0 < d < T(S_1, S_s)$ for loops with intermediate parallelism. In this model, each iteration is assigned to a virtual processor and execution of two successive iterations is delayed by $d$ time units. This is similar to cyclic scheduling discussed earlier.

In general, the delay $d$ can range from zero (the vector loop case) to $T$ (the sequential loop case), where $T$ is the execution time of one iteration of the loop. The total execution time for a DOACROSS loop $L$ with $N$ iterations for an unbounded number of processors is:

$$TE(L) = (N - 1) d + T$$  \hspace{1cm} (3.2)

When there are only $P$ processors, the execution time is [76]:

$$TEP(L) = \left\lceil \frac{N}{P} \right\rceil - 1 \max \{T, Pd\} + ((N - 1) \mod P) d + T$$  \hspace{1cm} (3.3)

In section 3.4, it was stated that data dependence could be either lexically-forward (data from higher indices used by iterations with lower indices) or lexically-backward (data from lower indices used by iteration with higher indices). Normally, lexically forward dependencies (LFD) do not contribute to delays in executing loop iterations. A lexically-backward dependence (LBD) can be transformed into a lexically-forward dependence by reordering the statements of the loop, provided the statements do not form a dependence cycle [28]. DOACROSS loops where the LBD cannot be transformed into LFD lead to delays in executing successive iterations. Hence, we focus our attention on regular DOACROSS loops with LBD.
Consider two statements of a loop, $S_4$ and $S_8$, where $S_4$ lexically precedes $S_8$. Statement $S_8$ of iteration $I_2$ computes the data used by statement $S_4$ of iteration $I_3$. The semantics of sequential programs guarantee that iteration $I_2$ is executed before iteration $I_3$. If these two iterations were assigned to different processors, a delay must be introduced before execution of iteration $I_3$, such that statement $S_8$ of iteration $I_2$ executes before statement $S_4$ of iteration $I_3$, in order to satisfy the dependence. Hence, a delay $d$ equal to at least 5 statements must be introduced to iteration $I_3$. This loop example exhibits a lexically-backward dependence.

The DOACROSS loop of the example shown in Figure 3.5 has $N = 8$ iterations, a delay $d = 4$, and a loop execution time $T = 10$. The parallel execution of the loop on 3 processors takes 38 time units, resulting in a speedup of 2.1.

Communication cost among processors is not included in this model. The overall execution time depends on the communication cost due to inter-iteration dependencies [92]. For a shared memory system, the delay $d$ should include not only the delay due to the lexically backward dependence (LBD), but also the delays in accessing shared variables. For distributed memory systems data must be shared using messages which take several orders of magnitude longer than a processor execution cycle. It has been reported that the Intel iPSC/1, iPSC/2, and iPSC/860 hypercubes have communication/execution ratios of 26, 59, and 1000 respectively, while the ratio for the nCUBE 3200 and 6400 hypercubes are 30 and 107, respectively [30]. Performance studies on CM-5 also show that some classes of problems are communication limited on that machine [58]. Using a balance factor $b = \frac{t_{\text{comm}}}{t_{\text{comp}}}$. $t_{\text{comm}} = t_{\text{total}} - t_{\text{comp}}$, a system is communication limited if $b \geq 1$. The Laplace solver on a 256 node partition has resulted in balance factors ranging from 2.11 for a 8192 x 8192 mesh size to 14.38 for a 64 x 64 mesh size. If we let $d$ be the delay due to the LBD, and $C$ be the total communication and synchronization cost (communication delay in the sequel) incurred, then the execution time of a DOACROSS loop with $N$ iterations on $P$ processors is:

$$ TEP(L) = \left( \left\lfloor \frac{N}{P} \right\rfloor - 1 \right) \max \{ T, P(d + C) \} + (\langle N - 1 \rangle \mod P) (d + C) + T \quad (3.4) $$
Figure 3.5. Allocation of a DOACROSS Loop.
Here for every delay $d$ due to LBD, a communication/synchronization cost $C$ is added, increasing the dependency delay from $d$ to $d + C$. For the example of Figure 3.5, if we assume a communication delay $C = 6$, the total parallel execution time becomes 80 time units, leading to essentially no speedup. Increasing or decreasing the number of processors will not change the execution time. Larger values for $C$ will make a parallel execution of the DOACROSS loop ineffective and lead to under-utilization of processors as they idle between the termination of an iteration and the initiation of the next assigned iteration.

Dynamic scheduling schemes such as GSS and Factoring are not effective in scheduling DOACROSS loops. When chunks (a number of consecutive iterations) are assigned to processors, iterations in successive chunks must wait for the completion of all iterations in the preceding chunks. Since chunk sizes are greater than one, the delay among processors assigned successive iterations is now equal to $(n - 1)T + d + C$, where $n$ is the size of the chunk assigned to the previous processor. The total execution time of the DOACROSS loop shown in Figure 3.5 using either GSS or Factoring is 56 when $C = 0$, and 80 when $C = 6$. Both schemes reduce the amount of communication overhead when compared to cyclic scheduling, at the expense of reduced parallelism. They perform worse than cyclic method when the communication cost is zero, but with non-zero communication cost, they perform no worse than the cyclic scheduling. The execution time for the same example using static chunking is 68 when $C = 0$, and 80 when $C = 6$. Thus, static chunking performs better when the communication cost is significant, since it only incurs $(P - 1)$ communication delays.

3.4.3.2. Pre-synchronized Scheduling (PSS)

Krothapalli and Sadayappan [55] proposed a dynamic scheduling scheme called pre-synchronized scheduling (PSS) for eliminating processor idle cycles that result from scheduling schemes such as GSS and Factoring. Here, iterations are scheduled only when their data dependencies and synchronization requirements are met. Loop iterations are uniquely identified using indices, and a ready queue of enabled iterations is maintained by a Global Control Unit (GCU). An idle processor gets an id from the ready queue and executes it. When the execution is complete, successor loop iterations that are enabled are added to the ready queue. A DOACROSS loop is divided into two separate loops and scheduled
separately. The two loops correspond to a $T - d$ portion of the loop that can be executed in parallel, and a $d$ portion that must wait for synchronization from previous iterations. This method introduces a scheduling/synchronization overhead equal to $2N$, resulting from the fact that we now have $2N$ loop iterations to schedule. The performance of PSS is largely dependent on two factors: (1) how the ready queue is implemented, e.g., FIFO, priority, and (2) the scheduling cost. Even though the $T - d$ portion is a parallel loop (akin to DOALL), it is necessary to schedule the iterations properly to facilitate an interleaved execution of iterations from the $T - d$ portion and the $d$ portion of a DOACROSS loop. For the example of Figure 3.5, the best performance achievable with PSS is 38, when the scheduling cost is ignored. This is comparable with that achieved by the Cyclic scheduling scheme. However, the PSS scheme incurs significant scheduling costs. We need to include a communication ($C$) each time an idle processor obtains the id of a loop iteration from the ready queue and a communication ($C$) to update the ready list when a processor completes a loop iteration. Since PSS schedules $2N$ loop iterations, we have a $4CN$ communication cost. For example, with $C = 6$, the execution time for the loop shown in Figure 3.5 will become 146. The cost can be reduced by assigning several loop id's (i.e., a chunk) each time an idle processor accesses the ready queue. However, it is difficult to arrive at an optimal chunk size.

3.4.3.3. Staggered Distribution Scheme

The Staggered Distribution scheme (SD) [47, 49, 65] was proposed to address the issue of detection and allocation of dynamic parallelism in a program. This scheme is based on an allocation policy which: i) detects dynamic parallelism for loop constructs at compile-time and, ii) allocates them to the estimated hardware resources in a staggered fashion using a set of heuristic rules. Here loop iterations are unevenly distributed among processors in order to mask the delay caused by data dependencies and communication. Performance studies have indicated that this scheme is effective for loops containing large degrees of parallelism among iterations. It has been observed that near optimal speedup can be attained even in the presence of communication delays [47].

In order to use SD for the DOACROSS loop, the loop is scheduled according to the following policy: the iterations assigned to $PE_i$ succeed the iterations assigned to $PE_{i+1}$, and $PE_i$ is assigned $m$ more
iterations than \( PE_{i-1} \). This results in more iterations assigned to higher numbered processors. For example, with 6 iterations and 3 processors we may assign 1, 2 and 3 iterations, respectively, to the 3 processors. The delay caused by iterations assigned to \( PE_{i-1} \) will be equal to \( d \) per iteration plus the communication cost \( C \). This delay will be masked out by the \( T - d \) portion of the additional iterations \( (m_i) \) assigned to \( PE_i \). The number of additional iterations assigned to \( PE_i \) is given by:

\[
m_i = \left\lfloor \frac{n_{i-1}d + C}{T - d} \right\rfloor
\]

(3.5)

where \( n_{i-1} \) is the number of iterations allocated to \( PE_{i-1} \), \( T \) is the execution time of one iteration, \( d \) is the delay and \( C \) is the inter-processor communication cost. The total number of iterations \( n_i \) allocated to \( PE_i \) would be:

\[
n_i = n_{i-1} + m_i = \left\lfloor \frac{n_{i-1}T + C}{T - d} \right\rfloor
\]

(3.6)

Thus, the Staggered distribution masks delays resulting from the lexically backward dependencies among loop iterations and the communication delays involved in transmitting the dependent data among processors since later processors execute more \( (T - d) \) loop iterations. The performance of this scheme can be fine tuned (to accommodate different communication costs) by selecting an appropriate number of iterations \( n_1 \) assigned to the first processor. Note that equation (3.6) also determines the maximum number of processors needed to execute the DOACROSS loop with \( N \) iterations. The synchronization overhead is only \( (P - 1) \times C \), which is smaller than that incurred by cyclic scheduling and PSS methods.

For the example shown in Figure 3.5, we use a distribution of 2-3-3 iterations when \( C = 0 \), giving 44 units of execution time and a speedup of 1.82. When \( C = 6 \) we use a distribution of 1-2-5 iterations, giving 50 units of execution and a speedup of 1.6. Staggered distribution accounts for different
communication costs by selecting an appropriate \( n_i \). The Staggered scheme however, distributes an uneven load among processors, heavily loading later processors.

### 3.4.4. Comparison of DOACROSS Scheduling Schemes

Table 3.3 compares and contrasts the characteristics of the three DOACROSS loop allocation schemes discussed. Cytron's cyclic scheduling scheme for DOACROSS loops does not take into consideration the effect of inter-processor communication cost. When the communication delays are significant, the overhead of this scheme increases as a function of \((n-1)*(C+d)\), and is independent of the number of PEs. This scheme offers low hardware utilization as a result of the processor idle cycles between the termination and initiation of successive iterations assigned to the same PE.

Pre-synchronized scheduling (PSS) attempts to balance the load and eliminate idle cycles, however, it introduces scheduling overhead proportional to \(2N\) and a communication cost of \(2C\) per iteration. The Staggered distribution scheme (SD) accounts for the processor delays due to LBD and communication. This is achieved by assigning a varying number of loop iterations to processors. This scheme achieves better results than the previous algorithms, and utilizes an optimal number of processors. The major weakness of the staggered scheme is the uneven load assignment to processors.

We have conducted simulation studies for determining the number of iterations assigned to processors at each scheduling step using the various schemes described. The results are shown in Table 3.4. Static chunking was included for the sake of completeness and because it performed better than the cyclic scheme when the communication cost is significant. The total execution time for SC shown in Table 3.4 was obtained by separating the loop into two separate loops as done in SD. Pre-synchronized scheduling (PSS) was not included in Table 3.4, since the authors have not suggested any strategies for managing the ready list, making an accurate analysis difficult. As discussed earlier, PSS in general
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic Scheduling (Cyclic) [28]</td>
<td>Exploits the parallelism present in a DOACROSS loop.</td>
<td>Does not take into consideration the effect of inter-processor communication cost. Overhead increases linearly as a function of ((n-1)*(C+d)). Offers low hardware utilization.</td>
</tr>
<tr>
<td>Pre-synchronized Scheduling (PSS) [55]</td>
<td>Balances the load and eliminate busy waiting period. Iterations are scheduled when their synchronization requirements are met.</td>
<td>Introduces scheduling overhead equal to (4CN). No implementation details on the ready queue management were presented. The performance of this scheme is unacceptable if the scheduling cost is significant.</td>
</tr>
<tr>
<td>Staggered Distribution Scheme (SD) [47, 49, 65]</td>
<td>Considers the effect of both delay ((d)) and communication cost((C)). Automatically controls and determines the maximum number of processors required for efficient execution of the loop based on the physical characteristics of the loop and the underlying machine architecture — higher resource utilization. Lowest scheduling overhead.</td>
<td>Produces an unbalanced load among processors, with the higher numbered processors receiving the larger amount of work.</td>
</tr>
</tbody>
</table>
TABLE 3.4. Number of iterations assigned to a processor at each scheduling step with $T = 10$, $d = 2$, $n = 500$, $C = 5$, $P = 4$.

<table>
<thead>
<tr>
<th>Step</th>
<th>PE</th>
<th>Cyclic</th>
<th>SC</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>125</td>
<td>85</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>125</td>
<td>108</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>125</td>
<td>136</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>125</td>
<td>171</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>27</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>31</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>34</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>35</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>36</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>38</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>39</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>43</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>44</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>47...500</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Total Execution Time: 3,503 2,015 1,710
performs poorly when the communication costs are significant. Table 3.4 shows that the cyclic scheme has the worst performance, followed by static chunking.

3.4.5. Performance Results of Loop Scheduling Schemes

There has been considerable interest in parallelizing loops since they are the major source of program parallelism. In this chapter, we examined how loops with inter-iteration dependencies (DOACROSS), and without dependencies (DOALL) can be executed in parallel. Both static and dynamic scheduling approaches were studied. The various approaches presented in this chapter were also compared for their complexity, scheduling overhead, communication cost, processor utilization, and expected speedup.

Yue and Lilja [103] measured the performance of the different DOALL scheduling algorithms on two different types of loops. The first loop is a matrix multiplication program which is parallelized on the outer loop. The size of the parallel tasks is large, and all the iterations have the same number of operations so that the variance in iteration execution times is small. The second loop is based on the adjoint-convolution process. It is parallelized on the outer loop and, in contrast to the first loop, each parallel iteration has a different number of operations so that it has a large variance in iteration execution times. The results are shown in Figures 3.6 and 3.7. The figures do not include the performance of self scheduling because it performs poorly on their system. The results of the first loop (Figure 3.6) show that all the algorithms performed similarly when $N$ is large and the variance is small. Hence, the effect of load imbalance is not significant. They also found that fixed-sized chunking performed better than the others when $N$ is small. On the other hand, the results of the second loop (Figure 3.7) show that if the variance is large, fixed-size chunking (FS) attains only half of the possible speedup. Guided Self-scheduling (GSS) also does not perform well as it assigns too much work at the beginning of the execution and does not save enough work at the end for balancing the load. Factoring and Trapezoid Self-scheduling (TSS) balance the workload better than the other schemes and attains significantly better speedup. It should be noted that when the number of iterations is small, none of the scheduling approaches perform well, since there
Figure 3.6. Performance of DOALL scheduling algorithms on matrix multiplication ($N = 300$).
Figure 3.7. Performance of DOALL scheduling algorithms on adjacent convolution (N = 100).
is insufficient work to offset the overhead due to scheduling and distribution of work. Based on these results, we can conclude that among the techniques investigated in this chapter for parallelizing iterations with varying execution times, fixed-size chunking performs well when the variations in execution times are small and the number of iterations are small, while factoring and TSS perform better when the variance is large. Unlike DOALL, iterations of DOACROSS loops must be executed in a predetermined order to maintain inter-iteration dependencies. As can be expected, the serialization of iterations leads to a significant loss of parallelism [19]. DOACROSS loops can be either regular or irregular. In a regular DOACROSS loop, inter-iteration dependence distances are constant. Staggered distribution scheme (SD) attempt to mask the communication delays resulting from inter-iteration dependencies. This is achieved by assigning monotonically increasing number of iterations to higher numbered processors. This scheme performs better than other scheduling methods for regular DOACROSS loops. Effectiveness of the Staggered scheme has been simulated and compared against those of static chunking (SC) and cyclic (CYC) scheduling [47, 49, 65]. Pre-synchronized scheduling was not considered, since the best-case performance of this scheme would be equivalent to cyclic scheduling.

Our test-bed includes a representative loop with the execution time of $T = 50$ and loops 3, 5, 11, 13, and 19 of the Livermore Loops, which have cross-iteration dependencies [32]: Loop 3 is the standard Inner Product function of linear algebra, Loop 5 is taken from a Tridiagonal Elimination routine, Loop 11 is a first sum, Loop 13 is a fragment from a 2-D Particle-in-Cell code, and Loop 19 is a general Linear Recurrence Equation. In this simulation:

1. The inter-PE communication delays are varied based on the ratio of communication time to iteration execution time ($C/T$).

2. Delays due to LBD are computed for various $k$ values, where $k$ is the fraction of delay $d$ to the execution time of an iteration $T$, $k = d/T$.

Figure 3.8 shows the maximum speedup attained by SD and CYC schemes for $n = 2000$, $C/T = 0.5$. The speedup for SD is significantly better than CYC for all cases. The Average Parallelism ($AP$) of the loop (can also be considered the maximum speedup of a loop) when $k = 0.1$ is equal to 9.9, which is very close to the speedup attained by SD even with communication overhead. The speedup for CYC is
Figure 3.8. Maximum speedup (MS), n = 2000, C/T = 0.5.
less than two for $C/T = 0.5$ and less than one when $k = 0.6$. Our simulation results also show that the maximum speedups attained by CYC for $C/T = 1.0$ and up are all less than one. This means that the loops can obtain better performance if they were executed serially in one PE. The number of PEs required to realize maximum speedup for CYC is shown in Figure 3.9. This number drops to two independent of $k$ for $C/T \geq 0.5$. This is due to the fact that for $C/T = 0.5$, after two iterations, the communication delay would be equivalent to the execution time of one iteration $T$. Therefore, the third and fourth iterations can be executed in the same two processors without any additional delay. The cycle will be repeated for every pair of iterations — Using more processors does not affect the performance.

Table 3.5 shows the speedup of SD over SC and CYC when the Livermore Loops were simulated. Timing values and inter-processor communication used in the simulation were based upon instruction and communication times for the nCUBE 3200 [30]. The ratio of communication to instruction execution ($C/E$) for the 3200 is 30. Loop 19 consists of two loops. Hence, each loop was tested separately (19(1) and 19(2)). The number of iterations for each loop were based on the specification of each loop. Loops 3, 5, and 13 were simulated for $n = 1000$, Loop 11 with $n = 500$, and Loops 19(1 & 2) with $n = 100$. Although the number of iterations for loops 11 can reach a maximum of 1000, we felt that 500 iterations would give a different perspective from Loop 3, since both loops (3 and 11) have the same value of $k$. The speedup for SD increases compared to SC and CYC as the $C/T$ ratio increases and decreases as the value of $k$ increases. There was not much speedup for Loop 13, since it had a negligible delay. For Loops 3, 5, 11, and 19(1 & 2) when $PE = 8$, the SD scheme utilized fewer PEs than the available number of PEs. These results show that SD offers better resource utilization. Furthermore, the number of PEs required also decreases as the communication cost increases.

The performance of the Staggered scheme has also been evaluated by running Loop 13 of the Livermore Loops on an nCUBE 2 multiprocessor. This scheme has been compared to Static chunking (SC) and Cyclic scheduling (CYC). Static chunking utilizes the same approach as staggered, i.e., separating the loop into two portions, and assigning equal number of iterations to each node/processor. Loop 13 was chosen due to its size and the fact that it has a large amount of exploitable parallelism ($AP = 4.29$). Furthermore, it possesses a reasonable amount of delay that hinders the ability of easily executing
Figure 3.9. Number of PEs to attain maximum speedup for Cyclic scheduling.
TABLE 3.5. Speedup of Staggered Distribution relative to Static Chunking Su(SC) and Cyclic Scheduling Su(CYC) for the Livermore Loops with C/E = 30. Actual number of PEs used by Staggered Distribution in parentheses.

<table>
<thead>
<tr>
<th>LOOP #</th>
<th>k</th>
<th>C/T</th>
<th>PE = 4</th>
<th>PE = 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Su(SC)</td>
<td>Su(CYC)</td>
</tr>
<tr>
<td>3</td>
<td>0.25</td>
<td>3.75</td>
<td>1.20</td>
<td>10.72</td>
</tr>
<tr>
<td>5</td>
<td>0.30</td>
<td>3.00</td>
<td>1.21</td>
<td>8.22</td>
</tr>
<tr>
<td>11</td>
<td>0.25</td>
<td>3.75</td>
<td>1.21</td>
<td>10.50</td>
</tr>
<tr>
<td>13</td>
<td>0.05</td>
<td>0.71</td>
<td>1.07</td>
<td>2.82</td>
</tr>
<tr>
<td>19(1)</td>
<td>0.33</td>
<td>3.33</td>
<td>1.24</td>
<td>7.53</td>
</tr>
<tr>
<td>19(2)</td>
<td>0.27</td>
<td>2.73</td>
<td>1.23</td>
<td>6.86</td>
</tr>
</tbody>
</table>

The loop in parallel. These two features together offer a very good balance between parallelism and delay that requires a very effective loop scheduling scheme to obtain a significant speedup.

The loop was run on the nCUBE 2 multiprocessor using 2 to 16 processors (PE). Figure 3.10 shows that the SD scheme again attained better speedup. Furthermore, the SD scheme utilizes less than 8 processors, since it controls the number of processors that are used effectively. The peak speedup for SD was 2.723 utilizing 7 PEs, which is a 36.5% speedup reduction from the average parallelism of 4.29, and SC had a 46.85% speedup reduction utilizing 8 PEs. Similar to the previous results, cyclic scheduling is ineffective if the communication cost is significant. The C/T ratio shows a significantly high rate of communication delay relative to the execution time of one iteration. The speedups attained by CYC for all PEs was 0.177.

3.5. Summary

In this chapter, we have shown the importance of partitioning programs into threads. A thread defines the granularity of a computation and thus the basic unit of work for scheduling. This granularity directly affects the amount of overhead required for synchronization and context switching. These
Figure 3.10. Speedup for loop 13 - nCUBE 2 system.
overheads are the parameters needed by the cache manager to enforce the prefetching and replacement policies. Hence, a partitioning algorithm should attempt to maximize parallelism while minimizing the overhead required to support the threads.

Once the program is partitioned into threads, the threads should be allocated or distributed among the processing elements in order to exploit the maximum parallelism in the program. We have also discussed about the allocation issue in dataflow projects, as well as other proposed schemes.

In the previous chapter, we have concluded that before cache can be effectively used in a dataflow environment, one has to find an effective means to detect and enhance localities in a dataflow program. As a result, the Vertically Layered (VL) allocation scheme [61], and the Staggered Distribution scheme (SD) [47, 49, 65], were discussed. Note that these schemes partition a dataflow graph into subgraphs, while preserving the localities, and distribute these subgraphs among processors in a multiprocessor environment.

We have shown how the VL allocation scheme can effectively detect spatial localities in a dataflow graph, by clustering nodes connected serially into a partition, and how the staggered distribution scheme preserves both spatial and temporal locality, by grouping together instructions belonging to the same iteration as well as successive iterations. These schemes also take into account the effect of inter-processor communication delays in determining the partitions. It would be interesting to see how effective it would be in performing this task, when these two schemes are combined together.

In the next chapter, we will present an extension to the Staggered distribution scheme, which will address the uneven load distribution problem.
4.1. Overview

In the previous chapter, we presented two schemes that partition a dataflow graph into subgraphs, while preserving the localities, and distribute these subgraphs among processors in a multiprocessor environment. These are the Vertically Layered (VL) allocation scheme, and the Staggered Distribution scheme (SD). These schemes were discussed as a means to detect and enhance localities in a dataflow program.

In this chapter, we propose an extension to the Staggered distribution scheme, called Cyclic Staggered (CSD), which addresses the uneven load distribution problem of the original staggered scheme. A performance comparison between CSD and the schemes previously reviewed in chapter 3, as well as the original staggered scheme, will be presented.

4.2. Cyclic Staggered Distribution (CSD)

A modified version of the staggered scheme called Cyclic Staggered (CSD) is proposed to address the uneven load distribution [48, 64, 44]. CSD will also handle loops with varying iteration execution times. The CSD has been found to be effective when the number of processors is less than those needed by the Staggered scheme (maxpe).

Unlike using $n_1$ iterations, CSD will start with one iteration assigned to the first processor, and $n_i$ iterations to the remaining $P-1$ processors based on equation (3.6) from chapter 3. The remaining iterations are redistributed to all $P$ processors based on the staggered allocation. Note that the delay that must be masked by higher numbered processors now is smaller than that in the original SD approach, since some loop iterations would already have completed due to prior staggered allocation. Thus a smaller number of
additional iterations are assigned to $PE_i$ as compared to equation (3.6). The number of iterations $n_i$ for this new scheme will be:

$$n_i = \left[ \frac{n_{i-1}T + C \cdot n_p T}{T - d} \right] = \left[ \frac{(n_{i-1} - n_p)T + C}{T - d} \right]$$ (4.1)

where $n_p$ is the number of iterations previously allocated to processor $PE_i$. This approach results in a more balanced load and improved speedup than the original staggered scheme on $P$ processors. When the execution times of loop iterations vary, CSD can use estimated worst case iteration execution time (possibly augmented by run-time support to adjust these estimates with actual execution times) in determining the distribution for the second and subsequent passes.

In order to compare CSD to the other schemes described in chapter 3, we obtained the information in Table 3.3 and included the features of CSD to produce Table 4.1. In the previous chapter, we stated that the major weakness of the staggered scheme is the uneven load assignment to processors. The Cyclic Staggered (CSD) answers this load imbalance of SD. It should be noted that the CSD results in larger communication delays for a loop than that with staggered scheme, however, the more balanced load of CSD leads to a better performance particularly when the number of processors is less than the optimal number required for SD [48, 64, 44].

4.3. Performance of Cyclic Staggered Distribution

To assess the effectiveness of this new scheme, we again conducted simulation studies for determining the number of iterations assigned to processors at each scheduling step using CSD and compared it to the other schemes described in chapter 3. The results are shown in Table 4.2. The table shows that the cyclic staggered scheme (CSD) produced the best performance.

Effectiveness of the Cyclic Staggered scheme (CSD) was also simulated and compared against the original Staggered scheme (SD) using the same test-bed as before. As can be seen in Figures 4.1 and
TABLE 4.1. Comparison of DOACROSS scheduling algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic Scheduling (Cyclic) [28]</td>
<td>Exploits the parallelism present in a DOACROSS loop.</td>
<td>Does not take into consideration the effect of inter-processor communication cost. Overhead increases linearly as a function of ((n-1)*(C+d)). Offers low hardware utilization.</td>
</tr>
<tr>
<td>Pre-synchronized Scheduling (PSS) [55]</td>
<td>Balances the load and eliminates busy waiting period. Iterations are scheduled when their synchronization requirements are met</td>
<td>Introduces scheduling overhead equal to (4CN). No implementation details on the ready queue management were presented. The performance of this scheme is unacceptable if the scheduling cost is significant.</td>
</tr>
<tr>
<td>Staggered Distribution Scheme (SD) [47, 49, 65]</td>
<td>Considers the effect of both delay ((d)) and communication cost ((C)). Automatically controls and determines the maximum number of processors required for efficient execution of the loop based on the physical characteristics of the loop and the underlying machine architecture — higher resource utilization. Lowest scheduling overhead.</td>
<td>Produces an unbalanced load among processors, with the higher numbered processors receiving the larger amount of work.</td>
</tr>
<tr>
<td>Cyclic Staggered Distribution (CSD) [48, 64, 44]</td>
<td>Balances the load by cyclically assigning the remaining iterations to processors, and at the same time masking out the effect of both delays due to LBD and communication. Increases the amount of communication delay, relative to the SD, but simulation results have shown that it still improves the performance and offers a higher speedup.</td>
<td>Advantages are only possible if the number of PEs available is less than (maxpe).</td>
</tr>
</tbody>
</table>
TABLE 4.2. Number of iterations assigned to a processor at each scheduling step with $T = 10$, $d = 2$, $n = 500$, $C = 5$, $P = 4$.

<table>
<thead>
<tr>
<th>Step</th>
<th>PE</th>
<th>Cyclic</th>
<th>SC</th>
<th>SD</th>
<th>CSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>125</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>125</td>
<td>108</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>125</td>
<td>136</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>125</td>
<td>171</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>27</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>31</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>34</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>35</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>36</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>38</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>39</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>43</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>44</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>46</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>47</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Execution Time | 3,503 | 2,015 | 1,710 | 1,298 |
4.2. CSD performed better than SD regardless of the values of \( n \), \( C/T \), and \( k \), especially when the number of PEs was halfway between 2 and \( \text{maxpe}-1 \). Finally, CSD attained an almost linear speedup for smaller number of PEs, even with delays due to LBD and communication cost. Since CSD outperforms SD, we can conclude that CSD comes even closer to the maximum speedup possible for a particular loop. However, these advantages are made possible only if the number of PEs available is less than \( \text{maxpe} \).

The performance of CSD has also been evaluated by running Loop 13 of the Livermore Loops on an nCUBE 2 multiprocessor. Figure 4.3 shows the speedup of the two staggered schemes. It was seen in Figure 3.10 that the number of PEs needed by SD to achieve maximum speedup (\( \text{maxpe} \)) was seven. Hence, in Figure 4.3, the number of PEs utilized for CSD is \( \text{maxpe}-1 \). Interestingly, unlike the previous results, CSD performed better than SD only when the number of processors is between 3 and 5. For an environment with two processors, SD outperformed CSD because of the additional overhead incurred to implement the cyclic staggered scheme — each PE has to continuously check for more iterations to execute after execution of each chunk. This overhead is also the reason for the small performance gain of the cyclic staggered schemes over SD compared to the previous results. SD does not perform these tests and incurs less communication delays. With these results in mind, we can conclude that the cyclic staggered scheme is also effective in the execution of DOACROSS loops.

4.4. Summary

In this chapter, an extension to the Staggered distribution scheme, called Cyclic Staggered (CSD), which addresses the uneven load distribution problem of the original staggered scheme, was introduced. A comparative analysis between CSD and the schemes previously reviewed in chapter 3, as well as the original staggered scheme, was presented.

It was shown that CSD performed better than SD, especially when the number of PEs was halfway between 2 and \( \text{maxpe}-1 \). CSD comes even closer to the maximum speedup possible for a particular loop, compared to SD. However, these advantages are made possible only if the number of PEs available is less than \( \text{maxpe} \).
Figure 4.1. Comparative analysis of the staggered schemes, $C/T = 3.0$. 

Speedup

Number of PEs

CSD : Cyclic Staggered Distribution
SD : Staggered Distribution Scheme

$k = 0.3$
Figure 4.2. Comparative analysis of the staggered schemes, $C/T = 5.0$. 

- CSD: Cyclic Staggered Distribution
- SD: Staggered Distribution Scheme

$k = 0.3$
Figure 4.3. Speedup of the staggered schemes for Loop 13 - nCUBE 2 system.
In the next chapter, we will demonstrate how we have incorporated the Vertically Layered Allocation Scheme (VL) and the two Staggered Schemes (SD and CSD) with some modifications, to develop a new allocation scheme that alleviates the shortcoming of the VL — no loop distribution.
CHAPTER 5
VL-STAG ALLOCATION SCHEME

5.1. Overview

In the previous chapter, we introduced an extension to the Staggered distribution scheme (SD), called Cyclic Staggered (CSD). This new scheme addresses the uneven load distribution problem of the original staggered scheme. It was shown that CSD performed better than SD and comes even closer to the maximum speedup possible for a particular loop.

In this chapter, we show how all these schemes (VL, SD, and CSD) can be combined together to develop a new allocation scheme, called the VL-Stag Allocation Scheme (VSAS). A performance comparison between VSAS and VL will be presented.

5.2. VL-Stag Allocation Scheme

A new allocation scheme is proposed, which incorporates the two Staggered Distribution schemes into the VL allocation scheme, we call it the VL-Stag Allocation Scheme (VSAS). This is done by first preprocessing the program graph and applying the Staggered Distribution schemes (SD and CSD) on loops. If the number of processors available is less than those needed by the Staggered (SD) scheme (maxpe), then Cyclic Staggered (CSD) will be used. The resulting iterations assigned to a PE are then grouped into one single node and assigned an execution cost equal to the sum of the execution cost of all these iterations. We will call these nodes iteration-nodes. Hence, the number of iteration-nodes would be equal to the number of PEs. The loops in the original program graph are then replaced by the resulting iteration-nodes. Furthermore, the parallel loop iterations (FORALL or DOALL loops) are equally distributed among the PEs and are also grouped into iteration-nodes. The VL allocation scheme, with the following modification, can then be applied: Preserve the distribution obtained from the Staggered scheme by disallowing more than one iteration nodes belonging to the same loop to be assigned to one PE.
5.3. Performance of VL-Stag Allocation Scheme

In this section, we report on the simulation of VSAS and its comparative performance analysis against the original VL allocation scheme. Our test-bed includes the program graphs obtained by translating IF1 graphs [91] — IF1 is an intermediate form generated from a SISAL compiler [17] — IF1 is an acyclic graph that defines dependencies among instructions in a program. Hence, they are an appropriate means to develop a conceptual framework for understanding program behavior and thus scheduling programs. This has also allowed us to use actual SISAL programs in our studies. The execution time for each node was obtained from the SISAL compiler's Execution Cost Profile. This is a machine dependent parameter, where each node/instruction is associated with number of clock cycles and is representative of modern processor execution costs. This parameter is used by the SISAL compiler to perform static partitioning as well as to decide if a loop is to be parallelized or serialized.

In our simulation, the Inter Processor Cost (IPC) was varied based on the ratio of communication time to instruction execution time — $C/E$. This was done in order to evaluate the performance of the algorithm for different architectural features. In [58], the data network latency of the CM-5 was found to vary from 2 - 70 microseconds, while the execution time of a single precision vector addition is approximately equal to 109 nanoseconds. Hence, the $C/E$ ratio of the CM-5 varies from 18 - 640. When active messages were used, the network latency only ranges from 2 - 7 microseconds. This means that in reality the $C/E$ ratio is in the range of 18 - 65. From this information, we varied the $C/E$ ratio of our experiments from 10 - 60.

In our study, Loop 19 of the Livermore Loops and Simple were chosen as representative program graphs. Loop 19 is a general Linear Recurrence Equation and consists of two subloops. It contains 26 nodes of which two are loop (compound) nodes. The first subloop contains 9 nodes and the second 11. When these are expanded according to the estimated number of iterations $N = 100$, the total number of nodes executed in the graph equals 2,024. Simple is a hydrodynamics and heat conduction code widely used as an application benchmark. It contains 115 nodes, with the first 114 nodes containing 12 loop nodes. The final node itself is a loop node containing 639 nodes (includes both regular and loop nodes).
Our experiments made use of the first 114 nodes, since it already contains a large amount of parallelism as well as several loops. When the loop nodes are expanded, the total number of nodes executed in the graph equals 5178. The execution time of nodes in both graphs ranges from 1 - 25.

Figures 5.1 shows the performance of the VL allocation scheme and VSAS for Loop 19 when the $C/E$ ratio is 10. The average parallelism or maximum speedup attainable in the subloops of Loop 19 are 2.94 and 1.98, respectively. The speedup attained by VSAS was very close to these maximum speedups even in the presence of IPC due to the fact that most of the remaining nodes of the program graph are sequential nodes. The maximum number of processors needed by VSAS to exploit the parallelism available was $maxpe = 5$, therefore no real benefit is gained by increasing processors beyond this maximum value. The speedup attained by VSAS is 1.75 using 2 processors and up to 1.91 using 5 processors. On the other hand, almost no gain was achieved using the VL allocation scheme resulting from serializing the two subloops and the fact that there was not really any parallelism available in the rest of the program graph. In fact the VL allocation scheme could not utilize more than 2 processors. When the $C/E$ ratio was increased to 50 (Figure 5.2), the speedups for VSAS decreased, but was still able to attain a speedup of 1.72 using 4 processors ($maxpe$). This is just a 9.95% decrease in speedup from the one attained using 5 processors with a $C/E$ ratio of 10. This further shows the capability of VSAS to exploit even the small amount of available parallelism in a program and in the presence of IPC.

The performance of these two schemes for the Simple benchmark is shown in Figures 5.3 and 5.4. More parallelism is available in Simple, hence VSAS was able to attain a higher speedup — A speedup of 5.6 was attained by VSAS with 16 processors, while the VL allocation scheme was only able to attain a speedup of 1.37 when the $C/E$ ratio is 10. When the $C/E$ ratio was increased to 50, the speedup for VSAS using the same amount of processors is 4.05, while almost no change was seen in VL. This illustrates the significant effect of communication overhead on the overall performance even when a large amount of parallelism exists. The speedups for VL only ranged from 1.32 - 1.37, since a large amount of
Figure 5.1. Speedup for Livermore Loop 19, C/E = 10.
Figure 5.2. Speedup for Livermore Loop 19, C/E = 50
Figure 5.3. Speedup for Simple, C/E = 10.
Figure 5.4. Speedup of Simple, C/E = 50.
the parallelism in the program graph was due to the execution on the loop iterations. This again shows the importance of loop distribution and the effectiveness of VSAS in performing this task.

5.4. Summary

In this chapter, a new allocation scheme that incorporates two very effective schemes — Vertically Layered (VL) allocation scheme and Staggered distribution schemes has been introduced. The staggered distribution scheme was incorporated into the VL allocation scheme in order to alleviate the shortcoming of the VL — no loop distribution. The staggered schemes have shown its effectiveness in distributing DOACROSS loops even in the presence of inter-processor communication cost. The effectiveness of the resulting allocation scheme, the \textit{VL-STAG Allocation Scheme} (VSAS), relative to the VL allocation scheme has been reported, based on simulation studies. VSAS attains significantly better speedup than VL in programs that have loops in them. In addition, it was shown that VSAS attains speedup close to the maximum speedup even for a small amount of loop parallelism and in the presence of inter-processor communication. Results further show the importance of loop distribution in the overall performance of programs.

In the next chapter, we present the feasibility of applying cache in a multithreaded dataflow environment. To exploit the effectiveness of cache in this environment, a scheme that detects localities in a dataflow graph, partitions the graph into subgraphs while preserving the localities, and distributes subgraphs among the processors in a multiprocessor environment is needed to reduce the amount of cache misses. We have found that the allocation scheme utilized in VSAS could very well be used in detecting this locality and partitioning. In VSAS, temporal and spatial locality are preserved by grouping together logically related threads — instructions belonging to the same iteration and successive iterations and at the same time effectively detects spatial localities in a dataflow graph by assigning nodes connected serially to a partition (a vertical layer). As a result, we utilize VSAS and propose a locality enhancing policy that preserves localities, which is essential in optimizing the effectiveness of cache in multithreaded dataflow architectures.
CHAPTER 6

VL-CACHE LOCALITY ENHANCING POLICY

6.1. Overview

In the previous chapter, a new allocation scheme called the *VL-STAG Allocation Scheme* (VSAS) was introduced. This scheme enhances the scope of the *Vertically Layered* (VL) allocation scheme by using the *Staggered* distribution scheme to alleviate the shortcoming of VL in handling loops. The goal is to detect localities in a dataflow program and hence to incorporate cache memory in a dataflow environment.

In this chapter, we propose a locality enhancing policy (VL-CACHE) that preserves locality, to optimize the effectiveness of cache in multithreaded dataflow architectures. The VL-CACHE takes advantage of VSAS. The proposed VL-CACHE has been simulated and its effectiveness has been analyzed.

6.2. Proposed Locality Enhancing Policy

Referring to our earlier discussion (Chapter 5), the VSAS algorithm assigns vertical layers to processors. Furthermore, in each processor, nodes in horizontal levels are rearranged in order to take into account the effect of communication cost — The execution of a node is delayed if it requires a result from a remote processor. Hence, the horizontal levels of execution would be different from the original horizontal levels derived by the VSAS algorithm. This could be done by determining the earliest starting time of each node (operation) with respect to the communication cost. The earliest starting time of an operation could then be used as a basis to determine the new horizontal levels.

In chapter 3, serially connected nodes that were assigned to a vertical layer were referred to as either a critical path or longest directed path (LDP). In this chapter, we will simply refer to them as threads. An example of threads that were allocated to a vertical layer/processor with their corresponding
horizontal layering \((h)\) is shown in Figure 6.1. The dependencies between threads were omitted in order to improve clarity. Nodes 1 to 8, 9 to 12, 13 to 16, and 17 to 22 represents threads A, B, C, and D, respectively.

Each thread is partitioned into groups of \(x\) nodes and these groups are interleaved and stored in memory — each partition is a multiple of the cache block size. In other words, starting from horizontal layer \(h = 1\), groups of \(x\) consecutive nodes from threads having nodes in \(h = 1\) are interleaved and assigned to memory. After exhausting all nodes in level \(h\), the algorithm repeats the same sequence of operations for unassigned nodes in level \(h + 1\). This sequence of operations will be repeated, exhaustively, until all the nodes in the graph are assigned to memory. Refer to Figure 6.1 with \(x = 4\), the first four nodes of thread A (nodes 1, 2, 3, and 4) are first assigned to the first 4 locations of memory. No other thread has a node in \(h = 1\), so \(h = h + 1 = 2\). The next thread that has an unassigned node in \(h = 2\) is thread B.

Therefore, nodes 9, 10, 11, and 12 are assigned to the next 4 locations in memory. Subsequently, the first four nodes of threads C (nodes 13, 14, 15, and 16) and D (nodes 17, 18, 19, and 20) would be allocated next to memory. At this point, all the nodes in \(h = 2\) are exhausted and hence \(h\) is incremented by 1. The next value of \(h\) with unassigned nodes would be \(h = 5\). Nodes 5, 6, 7, and 8 from thread A are allocated to the next 4 locations in memory. \(h\) is again incremented, since there are no more unassigned nodes in horizontal layer 5. At horizontal layer 6, only thread D has unassigned nodes (nodes 21 and 22) that are finally assigned to memory. The resulting node arrangement in memory is shown in Figure 6.2. This ordering policy assures that a cache block contains groups of nodes from each thread, once a cache line is loaded. This would establish locality based on simultaneity of execution. During execution time, instructions from independent consecutive blocks are interleaved to improve the degree of instruction level parallelism. In other words, according to the dataflow order of execution, the processor would alternately execute instructions from different threads that belong to the same horizontal layer.

The value of \(x\) is somehow dependent on the underlying computational paradigm and the scheduling policy. For example, in a dataflow environment, a small value of \(x\) should offer a high degree of instruction level parallelism by allowing more active threads in the cache. If on the other hand, the
Figure 6.1. An example of threads assigned to a vertical layer.
Figure 6.2. Memory allocation of Figure 6.1.
processor prioritizes the scheduling of all the instructions or a certain number of instructions from a sequential thread, then a value of \( x \) might be equal to the number of instructions the processor prioritizes from a thread. Here we assume that instructions and data have a one-to-one correspondence, therefore, once an instruction block is fetched and stored in cache memory (instruction cache), the corresponding data block is also stored in the operand cache. This increases the probability of a resultant token finding its destination frame location in the operand cache. This would also allow one to develop a realistic scheduling policy based on the memory reuse concept [52]. It should be noted that in E-level ordering [93], the nodes are also arranged based on horizontal layering, but the value of \( x \) is equal to one. In E-level ordering, the probability of having destination operands resident in the operand cache would be lower than the VL-CACHE policy. Hence, by having the value of \( x \) greater than one, we would have more destinations resident in operand cache from each thread. The value of \( x \) to optimize cache performance is determined from simulation based on the application program and physical characteristics of the underlying platform.

Our policy would also improve cache utilization, since it allows us to develop a look-ahead policy and prefetch future blocks into the cache. In addition, due to the dataflow nature of operations, it would allow us to develop a meaningful replacement policy — we always know which instructions are next in line to be executed and can therefore replace the ones that have been executed. For the data blocks, the size of the working set is a crucial factor to guarantee the availability of the resultant destinations in the cache. Small working sets would not allow this and hence introduces a cache miss.

6.3. Performance of VL-CACHE Policy

Our simulator was enhanced to measure the feasibility of the proposed locality enhancing policy (VL-CACHE). In addition, the simulator was extended to compare and contrast the VL-CACHE algorithm against E-level ordering. The simulator is capable of determining the cache miss ratio for various cache configurations. Our test-bed includes the program graphs obtained by translating IF1 graphs [91] — IF1 is an intermediate form generated from a SISAL compiler [17], an acyclic graph that defines dependencies among instructions in a program. Hence, they are an appropriate means to develop a conceptual
framework for understanding program behavior and thus scheduling programs. This has also allowed us to use actual SISAL programs in our studies.

The Fast Fourier Transform (FFT), Simple, and Sloop 74 were chosen as representative program graphs. Simple is a hydrodynamics and heat conduction code widely used as an application benchmark and Sloop 74 is a loop from Simple. Table 1 lists the characteristics of the test-bed programs used in our current experiment.

<table>
<thead>
<tr>
<th>Program</th>
<th>No. of Instruction References</th>
<th>No. of Operand References</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>173135</td>
<td>132522</td>
</tr>
<tr>
<td>Simple</td>
<td>227219</td>
<td>143919</td>
</tr>
<tr>
<td>SLoop 74</td>
<td>133210</td>
<td>90347</td>
</tr>
</tbody>
</table>

In order to compare the performance of VL-CACHE with E-level ordering, an attempt was made to determine the value of $x$. Sloop 74 was used for this experiment. Both the instruction and operand cache had an associativity = 2, and block size = 8. The size of the instruction cache was varied from 128 to 2048 bytes, while the operand cache was varied from 2 - 32 Kbytes. The larger operand cache size was due to the fact that loops could have several active instances competing for available operand cache resources, hence requiring more cache blocks to accommodate the activation frames for these iterations.

Figure 6.3 shows the performance of the instruction cache for both VL-CACHE and VL allocation policies based on different $x$ values. For VL, all the instructions in the first thread are first assigned to memory then the second thread and so on. Hence, in the case of VL, the size of $x$ varies depending on the thread length. We recorded a very low cache miss rate of 0.00015 at 128 bytes and 0.00013 for sizes 256-2048 bytes. It can be concluded that small instruction cache sizes could result in an exceptional performance.
Figure 6.3. Effect of different values of $x$ on instruction cache for Sloop74.
The performance of the operand cache is depicted in Figure 6.4. It shows that the best performance is attained when $x = 2$. In section 6.2, we stated that the value of $x$ is dependent on the type of scheduling implemented in the processor. Our model scheduled blocks according to the dataflow order of execution, and historically a smaller value of $x$ should allow one to exploit a higher degree of parallelism—nodes from the same horizontal layer or the next layer are the next instructions to be executed/enabled.

VL had shown the worst performance regardless of cache size. This is due to the fact that in VL a cache block, more or less, belongs to the very same thread and may not be referenced in a short time frame. As a result, it is possible that a cache block belonging to a running thread be swapped in and out several times. With these results in mind, the value of $x = 2$ is used in all of our succeeding experiments.

Figures 6.5 and 6.6 compare the performance of VL-cache to E-level ordering for Sloop74. In figure 6.5, similar to our earlier observation, there is no difference in the performance between VL-cache and E-level ordering for the instruction cache. On the other hand, with regard to the operand cache, VL-CACHE offered an improvement in performance over E-level (figure 6.6). This improvement in performance is attributed to the fact that VL-cache policy satisfies the non-far reaching effects of dataflow execution model. This means that the destinations of instructions are very close in terms of horizontal layers to the current instruction. Hence, by having the value of $x = 2$, VL-cache has more destinations resident in operand cache from each thread than E-level.

Figures 6.7 – 6.10 further show the improvement obtained by using VL-cache over E-level ordering on FFT and Simple. In these simulation runs, the instruction cache block size was varied to see the effect of having larger block sizes on miss ratio. Similar to our previous observations, regardless of the instruction cache block size both VL-cache and E-level ordering behaved the same (figures 6.7 & 6.9). However, as before, in the case of the operand cache, VL-cache offered a performance improvement over E-level policy (figures 6.8 & 6.10). Obviously, the miss ratio decreased as the operand cache size was increased. Finally, the performance improvement attained by using VL-cache over E-level ordering is shown in figures 6.11 and 6.12 for both FFT and Simple. In our experiments, we assumed that memory accesses required 6 cycles while cache accesses required 2 cycles. The largest improvement for FFT was
Figure 6.4. Effect of different values of $x$ on operand cache for Sloop74.
Figure 6.5. Performance of instruction cache using VL-cache and E-level ordering for Sloop74.
Figure 6.6. Performance of operand cache using VL-cache and E-level ordering for Sloop74.
Figure 6.7. Performance of instruction cache using VL-cache and E-level ordering for FFT.
Figure 6.8. Performance of operand cache using VL-cache and E-level ordering for FFT.
Figure 6.9. Performance of instruction cache using VL-cache and E-level ordering for Simple.
Figure 6.10. Performance of operand cache using VL-cache and E-level ordering for Simple.
Figure 6.11. Percentage improvement of VL-cache over E-level ordering for FFT.
Figure 6.12. Percentage improvement of VL-cache over E-level ordering for Simple.
5.26% when the operand cache size was 4K bytes, while for Simple it was 3.9% when the operand cache size was 2K bytes. The improvements tend to decrease as the cache size increases. VL-cache tends to perform well when the cache sizes are smaller. This feature is attractive for cases when the cache size of a processor is fixed or limited.

To further determine the effectiveness of the VL-cache scheme, our simulator was extended to allow prefetching. A simple prefetching scheme, wherein the processor fetches a fixed number of blocks adjacent to the fetched block was adopted. We varied the number of prefetched blocks from 1 to 6. Figures 6.13 – 6.15 show the prefetching effect on the performance of the operand cache for both the VL-cache and E-level ordering for various cache sizes. From figure 6.13, it can be concluded that for the 2K-byte operand cache, VL-cache ordering offers some improvement over E-level ordering. Here we see an almost constant gap between the miss ratio of VL-cache and E-level ordering. The miss ratio of VL-cache decreases somewhat when the number of prefetched blocks was increased from 1 to 3, and then starts to level off. The lowest miss ratio obtained, dropped by only 0.02 or 5% with prefetching compared to no prefetching. This demonstrates that performing prefetching obtains minimal improvement when the cache size is small for this type of application programs. Similar to our earlier observations (figure 6.4), in organizing cache blocks VL algorithm does not take non-far reaching effect of dataflow model into consideration. In other words, a cache block could be swapped back and forth between cache and main memory several times during the corresponding activation frame's lifetime. VL-cache showed an improvement of 10% over the best performance of E-level ordering, compared to only 6% without prefetching (figure 6.6). For the 8K-byte operand cache, the result is very similar, except that the amount of improvement has increased. The lowest miss ratio obtained, dropped by 0.025 or 17% with prefetching compared to no prefetching. A 23% improvement for VL-cache over E-level was obtained, compared to 12% without prefetching (figure 6.6). Finally, for the 32K-byte operand cache, an even greater improvement was obtained. The lowest miss ratio obtained, dropped by 0.057 or 64% with prefetching compared to no prefetching. A 45% improvement for VL-cache over E-level was obtained, compared to 10% without prefetching (figure 6.6).
Figure 6.13. Performance of operand cache with prefetching, using VL-cache and E-level ordering for Sloop74. Operand cache size = 2K bytes.
Figure 6.14. Performance of operand cache with prefetching, using VL-cache and E-level ordering for Sloop74. Operand cache size = 8K bytes.
Figure 6.15. Performance of operand cache with prefetching, using VL-cache and E-level ordering for Sloop74. Operand cache size = 32K bytes.
The performance of prefetching for Simple and FFT are shown in figures 6.16 and 6.17, respectively. For large applications like Simple, prefetching offers a significant improvement. For Simple (figure 6.16), the miss ratio for the 2K-byte cache, with 4 prefetched blocks is less than the miss ratio for the 8K-byte cache with no prefetching. In fact, this miss ratio is close to the miss ratio for 2 prefetched blocks for the 8K case. This means that prefetching allows similar performance for smaller cache sizes. The same results can be found for 8K-byte and 32K-byte cache sizes. The miss ratio of the 8K cache with 4 prefetched blocks performs better than the 32K cache with no prefetching, and with 5 prefetched blocks, the 8K case obtains the same result as the best result obtained for the 32K case. Again this shows that by using prefetching with VL-cache, for some applications, we can obtain the same performance using smaller caches than we would have with larger caches.

For FFT (figure 6.17), results similar to Simple were obtained between the 8K-byte operand cache and the 32K-byte operand cache. The miss ratio for the 8K cache with 3 prefetched blocks is lower than the miss ratio of the 32K cache with no prefetching. Also, the best performance for the 8K cache is very close to the best performance of the 32K cache.

6.4. Summary

In this chapter, a new locality enhancing policy called VL-cache, which utilizes the threads produced by the Vertically Layered Allocation Scheme has been introduced. This new scheme reorders instructions based on both horizontal and vertical layering to produce an arrangement in memory that enhances the performance of operand cache. The effectiveness of the enhancing policy (VL-cache), relative to E-level ordering has been reported, based on simulation studies. VL-cache attains better performance on operand caches than E-level ordering. In addition, VL-cache performs better when the operand cache size is small. Further observations show that VL-cache improves its performance even further when prefetching is performed. The performance of a smaller cache with prefetching is comparable to the performance of a much larger cache without prefetching. This shows the effectiveness of instruction reordering in improving the performance of cache.
Figure 6.16. Performance of operand cache with prefetching, using VL-cache for Simple.
Figure 6.17. Performance of operand cache with prefetching, using VL-cache for FFT.
CHAPTER 7
CONCLUSION

7.1. Summary

The main goal of this research was to study, develop, and analyze optimization techniques which
detect and enhance localities in a dataflow program with an eye to study the application of cache memory
in a dataflow environment. In order to achieve this goal, a set of techniques had to be developed and later
incorporated. These techniques also consist of previously developed effective schemes.

In chapter 2, the concept of dataflow was discussed. The evolution of dataflow architectures from
a pure dataflow model with specific hardware support, to a multithreaded dataflow model which
incorporates some control-flow methods into the dataflow approach, to alleviate some of the inefficiencies
with the pure dataflow method, has also been addressed. One particular area of interest is the incorporation
of cache in a dataflow environment.

Since localities within a program exploit the effectiveness of cache, we have shown that localities
can exist in dataflow programs as well. Hence, cache can potentially be effective in a dataflow
environment. However, before cache can be effectively used in a dataflow environment, several issues
must still be resolved. First, the detection and enhancement of localities in a dataflow program is essential.
Second, the limited amount of storage in a processor (cache) can severely limit the amount of latency that
can be tolerated. And lastly, appropriate fetching/prefetching and placement/replacement policies for
caches in a dataflow environment is needed. These policies have to be tailored to the dataflow execution
model in order for them to be effective. Use of statistical replacement algorithms for conventional caches,
such as least-recently used (LRU) are apt to cause misplacement, if used for instruction and operand
memories in a dataflow environment.

In chapter 3, we have shown the importance of partitioning programs into threads. A thread
defines the granularity of a computation and thus the basic unit of work for scheduling. This granularity
directly affects the amount of overhead required for synchronization and context switching. These
overheads are the parameters needed by the cache manager to enforce the prefetching and replacement policies. Hence, a partitioning algorithm should attempt to maximize parallelism while minimizing the overhead required to support the threads.

Once the program is partitioned into threads, the threads should be allocated or distributed among the processing elements in order to exploit the maximum parallelism in the program. We have also discussed about the allocation issue in dataflow projects, as well as other proposed schemes.

In chapter 2, we have concluded that before cache can be effectively used in a dataflow environment, one has to find an effective means to detect and enhance localities in a dataflow program. As a result, the Vertically Layered (VL) allocation scheme [61], and the Staggered Distribution scheme (SD) [47, 49, 65], were discussed.

We have shown how the VL allocation scheme can effectively detect spatial localities in a dataflow graph, by clustering nodes connected serially into a partition, and how the staggered distribution scheme preserves both spatial and temporal locality, by grouping together instructions belonging to the same iteration as well as successive iterations. These schemes also take into account the effect of inter-processor communication delays in determining the partitions.

In chapter 4, an extension to the Staggered distribution scheme, called Cyclic Staggered (CSD), which addresses the uneven load distribution problem of the original staggered scheme, was introduced. A comparative analysis between CSD and the schemes previously reviewed in chapter 3, as well as the original staggered scheme, was presented.

It was shown that CSD performed better than SD, especially when the number of PEs was halfway between 2 and maxpe-1. CSD comes even closer to the maximum speedup possible for a particular loop, compared to SD. However, these advantages are made possible only if the number of PEs available is less than maxpe.

In chapter 5, a new allocation scheme that incorporates two very effective schemes — Vertically Layered (VL) allocation scheme and Staggered distribution schemes was introduced. The staggered distribution scheme was incorporated into the VL allocation scheme in order to alleviate the shortcoming of VL — no loop distribution. The staggered schemes have shown its effectiveness in distributing DOACROSS loops even in the presence of inter-processor communication cost. The effectiveness of the
resulting allocation scheme, the VL-STAG Allocation Scheme (VSAS), relative to the VL allocation scheme has been reported, based on simulation studies. VSAS attains significantly better speedup than VL in programs that contain loops. In addition, it was shown that VSAS attains speedup close to the maximum speedup even for a small amount of loop parallelism and in the presence of inter-processor communication. Results further show the importance of loop distribution in the overall performance of programs.

In chapter 6, a new locality enhancing policy called VL-cache, which utilizes the threads produced by the Vertically Layered Allocation Scheme was introduced. This new scheme reorders instructions based on both horizontal and vertical layering to produce an arrangement in memory that enhances the performance of operand cache. The effectiveness of the enhancing policy (VL-cache), relative to E-level ordering has been reported, based on simulation studies. VL-cache attains better performance on operand caches than E-level ordering. In addition, VL-cache performs better when the operand cache size is small. Further observations show that VL-cache improves its performance even further when prefetching is performed. The performance of a smaller cache with prefetching is comparable to the performance of a much larger cache without prefetching. This shows the effectiveness of instruction reordering in improving the performance of cache.

7.2. Future Directions

Our encouraging results with VL-cache have led us to believe that this scheme can be applied to a variety of architectures. The reordering strategy (size of $x$) must be tailored to the type of scheduling implemented in the processor. If scheduling is based on the dataflow order of execution as in our simulation studies, a value of $x = 2$ was found to be optimal. If the processor prioritizes the scheduling of all the instructions or a certain number of instructions from a sequential thread, then a value of $x$ might be equal to the number of instructions the processor prioritizes from a thread.

We are currently enhancing the ETS cache simulator to be able to perform smart prefetching and replacement of cache blocks, in order to further reduce the amount of cache misses. Smart replacement policies improve cache utilization, since we always know which instructions are next in line to be executed and can therefore replace the ones that have been executed. Prefetching, on the other hand, can also reduce
memory latency, since we can predict which instructions are going to be executed next and prefetch these blocks into cache before they are needed. We are also looking into introducing process control to reduce the number of activation frames competing for the available cache resources [53], especially for loops where a large number of iterations are active at the same time. Finally, we would like to incorporate cache memory reuse [52] into our scheme to reduce the amount of operand cache misses, by reusing or reassigning used operand locations in operand cache to succeeding instructions. This eliminates the need to fetch an operand cache block into memory, since the destination location is already in cache. Results from this research are very encouraging and thus will lead us to conduct further research into this area.
REFERENCES


VITA

Joford Lim is a PhD candidate in the Computer Science and Engineering department at the Pennsylvania State University in University Park, PA. He also received his M.S. degree in Computer Engineering in 1993 from the Pennsylvania State University, where he wrote Partitioning Algorithm for Dynamic Dataflow Machines for his Masters Thesis. His doctoral research interests were in loop scheduling, program allocation for multiprocessors, locality enhancing algorithms to improve the performance of caches, and implementation of cache in a multithreaded dataflow environment.