A Compiler Algorithm for Scheduling Non-Numeric Code
with Explicitly Forwarded Data on a Speculative Multiprocessor

by

Antonia Bingheng Zhai

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

© Copyright by Antonia Bingheng Zhai 1998
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-40959-7
Abstract

A Compiler Algorithm for Scheduling Non-Numeric Code with Explicitly Forwarded data on a Speculative Multiprocessor

Antonia Bingheng Zhai
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
1998

Thread-Level Data Speculation (TLDS) aim to improve the performance of non-numeric applications by parallelize likely independent threads. When frequently occurring thread-carried data dependences exist, the sources of the data dependences are explicitly forwarded between threads. With forwarding, the performance of the multi-processor is determined by the critical forwarding path length, which is equal to the non-overlapping portions of the thread. The compiler can schedule instructions to minimize the critical forwarding path length, and improve the performance of the parallel execution. This thesis presents a scheduling algorithm that identifies the frequently occurring thread-carried data dependences and moves the sources of these data dependences as early as possible in the thread. This scheduling algorithm aggressively moves the identified operations and computations that they depend on across a large number of instructions. This algorithm is able to exploit parallelism between threads, hence significantly improve the performance of the applications on TLDS architecture.
Acknowledgements

The research presented in this thesis would not have been possible without the help and encouragement from many others.

First and foremost, I must thank my supervisor Todd Mowry for his efforts in this work, for his guidance and encouragement. Especially for the many helpful comments on the organization and presentation of this work.

I would also like to thank Tarek S. Abdelrahman, Charles Clarke and Jimmy Xu for serving on my thesis committee.

I would also like to thank my fellow students at the University of Toronto for contributing to an enjoyable graduate experience.

In addition, I wish to gratefully acknowledge the financial support provided by NSERC for this research.
# Contents

1 Introduction .......................... 1
   1.1 Thread-Level Data Speculation (TLDS) ................................... 2
   1.2 Related Work .................................. 6
   1.3 Objectives and Organization .................................. 8

2 Scheduling Algorithm .................. 10
   2.1 Definitions .................................. 10
   2.2 Program Representation .......................... 12
      2.2.1 Control Flow Graph .......................... 12
      2.2.2 Def-Use Chains .......................... 12
      2.2.3 Static Single Assignment Form .......................... 13
      2.2.4 Data Flow Graph .......................... 14
   2.3 Problem Formulation and Optimization Constraints .................. 16
      2.3.1 Control Dependence Constraints .......................... 16
      2.3.2 Data Dependence Constraints .......................... 20
   2.4 The Complete Algorithm .......................... 21
      2.4.1 Initialization .......................... 21
      2.4.2 Scheduling Algorithm .......................... 24
   2.5 Examples .................................. 28
      2.5.1 Regions Without Inner Loops .......................... 28
      2.5.2 Regions With Inner-Loops .......................... 34
      2.5.3 Loop Splitting .......................... 34

3 Algorithm Evaluation .................. 41
   3.1 Implementation Framework .......................... 41
3.2 Evaluation Methodology ........................................ 42
   3.2.1 Performance Estimation .................................. 44
   3.2.2 Accuracy of Evaluation .................................. 46
3.3 Results ......................................................... 47

4 Conclusions ....................................................... 51
   4.1 Future Directions ........................................... 52
List of Tables

3.1 Benchmark characteristics ........................................ 48
3.2 Performance improvement through instruction scheduling .......... 49
List of Figures

1.1 Examples of data speculation ........................................... 2
1.2 Critical path in a speculative region ................................. 4
1.3 Coarse-grain synchronization vs. fine-grain synchronization for forwarded data .................................................. 5
1.4 Potential region speedups on four processors assuming a 10-cycle forwarding latency, 32B cache lines, and accounting for recovery time (\(F =\) forwarding with fine-grain synchronization, \(S =\) forwarding with fine-grain synchronization and aggressive instruction scheduling) .................................................. 6

2.1 SSA form does not preserve enough control flow information .......... 14
2.2 The program re-visited using DFG representation ...................... 15
2.3 Augment the DFG to identify no-write paths ............................ 18
2.4 One side of the branch is favored ....................................... 19
2.5 Both sides of the branch are equally likely ............................. 19
2.6 Data dependences form a cycle .......................................... 21
2.7 Algorithm for computing path length of each basic block of a region. ... 23
2.8 Typical data profiling information ....................................... 25
2.9 Algorithm for collecting related reads and writes to form sets .......... 26
2.10 The overall code motion algorithm ...................................... 27
2.11 Function moveup checks if the code motion violates any data dependence. 29
2.12 A loop from go: a region with four levels of if-branches ............. 30
2.13 The CFG and the DFG of the region with four levels of if-branches ... 31
2.14 The CFG of the region with four levels of if-branches after instruction scheduling .................................................. 32
2.15 A region with four levels of if-branches after code scheduling. (\(\Diamond\) marks the forwarding instruction) .......................... 33
2.16 The main loop in compress contains a back edge. ................................ 35
2.17 The CFG and DFG of the main loop in compress, it contains a back edge. 36
2.18 The main loop of compress after instruction scheduling .......................... 37
2.19 A loop from espresso, loop fission is needed to move part of the loop back. 38
2.20 Loop scheduling that requires loop fission ............................................. 39
2.21 A loop from espresso, the loop is split into two parts. Part of the loop is moved to the beginning of the region. (◇ marks the forwarding instruction) 40

3.1 Compiler Framework .................................................................................. 43
3.2 Execution time estimation with different forwarding path lengths ............. 45
3.3 Code motion increases register pressure ..................................................... 47
3.4 Potential regional speedups on four processors (P = No critical forwarding path minimization, S = Using the critical forwarding path minimization algorithm to perform code motion, R = Including the cost of recovery operation from failed speculation) ...................................................... 50
Chapter 1

Introduction

High-performance, general-purpose microprocessors serve as cost-effective computing engines for a wide spectrum of computers, ranging from personal computers to large-scale parallel computers. The wide variety of applications that run on microprocessors has placed an ever-increasing demand on the performance of such devices. This trend has motivated hardware and compiler designers to investigate increasingly aggressive techniques to exploit parallelism in applications.

Instruction-level parallelism (ILP) exploits independence among the instructions in a single thread of execution by issuing multiple instructions per cycle. Although parallel execution of those independent instructions has been used to improve performance in the past, ILP alone is limited because there is only one thread of control in this computational model. Thread-level parallelism (TLP) can be used to overcome this limitation by allowing multiple threads of execution to fetch and execute instructions from different locations of an application [10]. Since the coarse-grained TLP is largely orthogonal to the fine-grained ILP, it may be possible to exploit both forms of parallelism in a complementary fashion.

To fully exploit the potential of thread-level parallelism, it is necessary to extract independent threads of execution that can run in parallel from the application. Since it is unrealistic to expect programmers to write only parallel programs, we prefer to be able to automatically parallelize programs using a compiler. To date, automatic parallelization has only been successful with numeric applications that have regular memory access patterns [6, 12]. Unfortunately, for a large number of non-numeric applications little parallelism has been extracted by the compiler. These non-numeric applications are
characterized by complicated control flow and ambiguous data dependences that are difficult, if not impossible, to statically predict at compile time. Fortunately, with proper architectural support, we can relax these constraints on the compiler, by allowing it to execute threads that are likely to be independent in parallel, and by dynamically verifying the correctness of this execution at run-time. One example of an architecture that gives the compiler this flexibility is called Thread-Level Data Speculation (TLDS). A number of speculative concurrent multi-threaded execution models [4, 11, 18] have been proposed.

### 1.1 Thread-Level Data Speculation (TLDS)

To maximize parallelism, we would like to perform load operations as early as possible, thereby allowing operations that depend on them to also be executed early. However, preserving program semantics imposes restrictions on the order in which memory accesses can occur. A memory data dependence occurs when a memory location is accessed by two statements, and at least one of them is a write. Since the memory access patterns of non-numeric applications are often highly dependent on runtime data, it is difficult for the compiler to precisely predict memory data dependences for these codes. To improve performance despite this constraint, one attractive option is to speculatively move a read ahead of a write (if the two are unlikely to refer to the same memory address) and to resolve whether this is safe at run-time. This technique is known as data speculation.

Figure 1.1(a) shows a sample code fragment, where the compiler is uncertain about whether pointers p and q refer to the same address. Figure 1.1(b) shows the code fragment under instruction-level data speculation, while Figure 1.1(c) shows the fragment of codes parallelized under TLDS. In both cases, the load is speculatively moved ahead of the store.
and the safety of this speculation is dynamically verified at run-time [19]. Specifically for the case of thread level parallelization, the load is safe provided that its memory location is not subsequently modified by a store that belongs to an earlier thread.

In the case of TLDS, we are only concerned with thread-carried data dependences, where the source and destination of the data dependences are executed by separate threads of control that run in parallel. Since dependences with source and destination in the same thread are executed in the same sequential order, as they would be with an uniprocessor under TLDS, sequential order is preserved by default.

There are three types of data dependences: write-after-write (WAW), write-after-read (WAR), and read-after-write (RAW). It is well known that of two types of dependences, both write-after-write and write-after-read dependences (which are often referred to as output and anti dependences, respectively) can be eliminated through renaming. The TLDS hardware accomplishes this renaming by effectively using the individual primary caches as rename buffers for speculatively modified data [20]. In contrast, read-after-write (RAW) dependences are true data dependences, and they cannot be eliminated. Therefore the focus of this study is how to minimize the performance degradation caused by these true dependences. Throughout the remainder of this thesis, we will use the expression "data dependence" to refer to true (i.e. RAW) data dependences unless otherwise specified.

Data dependence violations result in recovery actions that can potentially degrade performance. Therefore, thread-level data speculation allows multiple threads to be executed concurrently with the hope that data dependences do not occur frequently between consecutive threads. When data dependences occur frequently, it is preferable to synchronize the threads by explicit forwarding the data from the producer to the consumer thread, because stalling a thread to wait for explicit synchronization is less expensive (from a performance point of view) than frequently invoking the recovery mechanism. The performance of the speculative region that requires forwarding is limited by the critical path length, which is the maxim non-overlapped portion of each thread plus the forwarding latency. Figure 1.2 shows the critical path of a speculatively executed loop, where each iteration of the loop is executed on a separate thread. Note that, in this example, the computation done in the critical path is not inductive, and therefore cannot be eliminated.
do {
    Wait A;
    Load A;
    ...
    if(condition(A)) {
        A = 1;
        Store A;
    } else {
        A = 2;
        Store B;
    }
    Signal A;
} while(1)

Figure 1.2: Critical path in a speculative region
By minimizing the critical path length, we can significantly improve the performance of speculative execution. Steffan and Mowry [20] observed that code scheduling has a large impact on the performance of many speculatively executed regions. Moving as many instructions out of the non-overlapped portion of a thread as possible may significantly reduce the critical forwarding path, hence improving performance.

When multiple values are to be forwarded, the synchronization can occur at either a coarse granularity (where forwarding is done once per thread) or at a fine granularity (where forwarding is done once per value). As illustrated in Figure 1.3(a), computing the critical path length is straightforward for coarse-grained synchronization. With fine-grained synchronization, as illustrated in Figure 1.3(b), there are multiple forwarding paths. The critical path length is the longest of all of the forwarding paths, which could be significantly smaller than that with coarse-grain synchronization.

Steffan and Mowry [20] demonstrated the performance benefit of combining fine-grain synchronization with aggressive instruction scheduling through a detailed simulation of
Figure 1.4: Potential region speedups on four processors assuming a 10-cycle forwarding latency, 32B cache lines, and accounting for recovery time ($F =$ forwarding with fine-grain synchronization, $S =$ forwarding with fine-grain synchronization and aggressive instruction scheduling).

ten regions parallelized under TLDS where forwarding is required. For the sake of completeness, we reproduce some results from Steffan and Mowry’s study in Figure 1.4. The simulation is done assuming four processors. Case $F$ includes optimizations such as eliminating induction variables and reduction operations, using parallel library routines, eliminating dependences due to forwardable scalars, and also performing fine-grained synchronization to forward scalar values. Case $S$ includes the same optimizations, but also uses instruction scheduling to maximize parallel overlap. By comparing case $F$ and case $S$ in Figure 1.4, we see that in seven out of the ten regions code scheduling alone can improve performance by 20% to 400%. Three regions achieve a regional speedup of roughly twofold or more relative to the no instruction scheduling case.

1.2 Related Work

Having recognized the performance impact of forwarding on a speculative multiprocessor, other research groups have also developed compiler techniques to minimize the length of
the forwarding path.

The Multiscalar architecture [18] uses a centralized Address Resolution Buffer (ARB) [5] to hold the results of the speculative memory operations of currently active threads, which can be used to detect memory data dependence violations [13]. However, synchronization is not scheduled between frequently occurring data dependencies, and the compiler does not optimize the code ordering to minimize the performance degradation caused by inter-thread memory data dependence violations. Compiler controlled synchronization only occurs between register communication, and this type of communication is most amenable to analysis. Registers are divided into two sets called "ModSet" and "UnModSet" at compile time. Each represents the set of registers that may be modified and the set of registers that are guaranteed not to be modified, respectively. A thread propagates the value in UnModSet to the next thread directly at the beginning of its execution. However, the value in the ModSet are forwarded with specific instructions (when they become available). These forwarding instructions are specified by the compiler.

The Multiscalar compiler uses a simple code scheduling scheme to minimize the length of the forwarding path. This scheme moves instructions in the control flow graph one block at a time [22]. Moving a producer instruction earlier in a thread also requires that the scheduler move any instructions that it depends upon; similarly, moving a consumer instruction later in a thread also requires the scheduler to move any instructions that are dependent on it. To accomplish this goal, both backward and forward program slicing [17, 14] are used to compute the computation chain related to a certain instruction. This is not a systematic way to approach the problem, and it has several disadvantages. First, it neither optimizes the most frequently executed path, nor allows the compiler to move instructions speculatively across control branches. Second, control branches are not moved, even when moving the control branches is beneficial to performance. Finally (and most important of all), only register communication is optimized with code motion, since register-allocated scalars are the easiest to analyze.

The Superthreaded architecture [21] uses a different approach to forward values between consecutive dependent threads. In the Superthreaded architecture, execution is partitioned into stages: (i) a continuation stage, (ii) a target-store-address-generation(TSAG) stage, (iii) a computation stage, and (iv) a write-back stage. The continuation stage computes induction variables and forks the next thread. The TSAG stage computes the
addresses of store operations upon which the successor threads may depend. These addresses, referred to as *target store addresses*, are then forwarded to the memory buffers of the successor thread. When a thread executes a load operation during the computation stage, if the address of which matches one of the target store addresses in the memory buffer, the thread will stall until the value becomes available [21].

Loop transformation techniques such as *loop shifting* [2] could convert some inter-thread data dependence to intra-thread data dependence, but it does not work for general cases. A forward dependence has a source located closer to the top of the region, and a destination located closer to the bottom of the region. A forward data dependence can either be *inter-thread* or *intra-thread*. Similarly, a backward dependence has a source that is closer to the bottom of the region, and a destination that is closer to the top of the region. Therefore, it must be an inter-thread dependence. A backward data dependence usually creates a long critical forwarding path. When a loop is shifted, the loop must be *cut* at some point, and that point becomes the new head of the loop. All data dependence edges that are *cut* reverse directions (that is, forward dependences become backward, and backward dependences become forward). It is not always possible to find a place to cut the loop where only backward dependences are present.

The scheduling algorithms discussed above do not perform instruction scheduling as aggressively as necessary for our purpose. What we need is a systematic approach that schedules forwardings as early as possible in a thread, so that the thread-carried data dependences in later threads can be resolved early. We also have to schedule the computations that those forwardings depend on as early as possible in the thread. With control profiling information, the compiler may even speculatively move the computation of forwarding values ahead of the control branches on which it depends on if it is on a frequently executed path.

### 1.3 Objectives and Organization

In this thesis, we present an algorithm that takes the sources and sinks of the frequently occurring data dependences and minimizes the resulting forwarding path using code scheduling. Specifically, we describe a static scheduler that schedules the forwarding of values for thread-carried data dependences, reducing the amount of time the consecutive
thread has to stall to wait for these values. Moving the source of a dependence in the static code often requires moving the computations that it depends on.

To maximize performance, we perform aggressive code scheduling on the most frequently executed path of a thread by allowing the compiler to speculatively move instructions ahead of the control branches upon which these instructions depend. When the control dependence is finally resolved at run-time, if the wrong instructions have been speculatively executed and the wrong value has been forwarded, then recovery operations are performed. This code motion is beneficial as long as the performance gain obtained on the most frequently executed path exceeds the recovery cost.

The remainder of this thesis is organized as follows. Chapter 2 presents the compiler analysis and code motion algorithm. Chapter 3 demonstrates the potential performance benefit of the algorithm using several loops found in the SPEC95 and SPEC92 benchmark suites. Finally, chapter 4 presents conclusions and future work.
Chapter 2

Scheduling Algorithm

Under TLDS, we have observed that code scheduling has a significant impact on the critical forwarding path length, which determines the performance of parallel execution with frequent thread-carried data dependence. This chapter presents an algorithm that uses the dependence information obtained through profiling to schedule instructions to improve performance of TLDS. This chapter is organized as follows. First, we define terminology used to describe the algorithm. Then we discuss some special cases that our algorithm has to take into account. Finally, we present the entire algorithm and provide several examples to demonstrate different aspects of the algorithm.

2.1 Definitions

Before describing the scheduling algorithm, we define some terminology that we will use later in this chapter. Note that many of these definitions are for concepts that have been defined elsewhere and are presented here for completeness.

Definition 2.1 A Control flow graph (CFG) \([1] G = (V, E)\) is a directed graph in which nodes represent statements, and an edge \(v \rightarrow w\) represents possible flow of control from \(v\) to \(w\). Set \(V\) contains distinguished nodes START and END such that every node is reachable from START, and END is reachable from every node in \(G\). START has no predecessors and END has no successors.

Definition 2.2 Node \(w\) is said to dominate node \(v\) if every path from START to \(v\) contains \(w\).
Definition 2.3 Node $w$ is said to post-dominate node $v$ if every path from $v$ to END contains $w$.

The dominance/post-dominance relationship is transitive, and the transitive reduction is a tree-structured relation called the dominance/post-dominance tree.

The definition of post-dominance leads us to the concept of control dependence [16]. Let $C$ denote the set of all pairs $(e, w)$ such that node $w$ is control dependent on edge $e$.

Definition 2.4 A node $w$ is said to be control dependent on edge $u \rightarrow v \in E$ if

- $w$ post-dominates $v$, and
- if $w \neq v$, then $w$ does not post-dominate $u$.

Definition 2.5 $cd(e) = \{w \in V|(e, w) \in C\}$ is the set of nodes that are control dependent on edge $e$.

Definition 2.6 $conds(w) = \{e \in E|(e, w) \in C\}$ is the set of edges on which node $w$ is control dependent.

Definition 2.7 $cdequiv(w) = \{v \in V|conds(v) = conds(w)\}$, is the set of nodes that have the same control dependences as node $w$.

Data flow analysis is a process of collecting correlated information in a program. Information collected at various points in a program can be related using simple set equations [1]. Dense data flow analysis problems require information to be propagated to the entrance and exit of every basic block. On the contrary, sparse data flow analysis problems only require information to be propagated to where the relevant data are used or modified. Johnson and Pingali [7] define single entry and single exit (SESE) region to help accelerate the analysis for sparse type data flow analysis problems. By identifying those SESE regions that do not contain useful control or data dependence information, we may avoid propagating information through these regions, thereby improving the data flow analysis efficiency.

Definition 2.8 An SESE region (Single Entry Single Exit) in a graph $G$ is an ordered edge pair $(a, b)$ of distinct control flow edges $a$ and $b$ where
- *a dominates b*,
- *b post-dominates a*, and
- *every cycle containing a also contains b and vice versa.*

**Definition 2.9** A Strongly Connected Component (SCC) in a directed graph is a maximal subgraph N where each node has a directed path to any other nodes in the same subgraph.

### 2.2 Program Representation

In order to schedule forwarding early in a thread, our algorithm searches for a chain of computations that leads to the value of a forwarding variable within a large number of instructions. Then all instructions on this chain are scheduled. In searching for this chain of computations, we would like to propagate information directly from the definition of a data object to where it is used without losing control dependence information.

Compilers may use a variety of different representations to capture the structure of a program, and the choice of representations determines what kinds of analysis the compiler can perform efficiently. We look for representations that can be used to solve sparse data flow problems efficiently, and support both forward and backward analysis.

#### 2.2.1 Control Flow Graph

A Control flow graph (CFG) is the conventional way to represent the structure of a program. We can perform dense data flow analysis [1] on a CFG. Dense data flow analysis allows us to solve both forward and backward data flow problems such as reaching definition and dead code elimination. Performing data flow analysis on a CFG suits dense data flow problems better, since information is evaluated at the entrance and exit of every basic block regardless of whether the basic block contributes any useful information.

#### 2.2.2 Def-Use Chains

Def-Use chains [23] is the first attempt at a program representation that allows information to propagate directly from definitions to uses. A Def-Use chain is defined as
follow:

**Definition 2.10** A Def-Use chain for a variable $x$ corresponds to an edge pair $(e_1, e_2)$ that satisfies the following conditions:

1. the source of $e_1$ defines $x$,
2. the destination of $e_2$ uses $x$
3. there is a control flow path from $e_1$ to $e_2$, which contains no assignments to $x$.

Unfortunately, this representation has two disadvantages:

- It is inefficient in the sense that the worst case complexity of this representation is $O(E^2V)$, where $E$ is the number of edges and $V$ is the number of nodes in a CFG.
- It lacks control dependence information, and therefore it cannot be used in backward data-flow analysis.

### 2.2.3 Static Single Assignment Form

Static Single Assignment (SSA) form [3] is designed to represent both the control flow and data flow properties of a program. In SSA form, each lexically distinct assignment of a variable is given a unique name, and all uses of this variable are renamed to match the new assignments. Since every use of a variable may be reachable from several definitions, SSA form uses join nodes at where multiple definitions meet, and $\phi$ functions are inserted at these nodes. $\phi$ functions are of the form:

$$V_n = \phi(V_1, V_2, \ldots) \quad (2.1)$$

where each operand represents an assignment of variable $V$ that may reach the join node through one of its incoming edges. The head of an edge in SSA form is an assignment to the variable, and the tail is a use of the variable. A $\phi$ function could be the head or the tail of an edge. The edges in SSA are defined as follows:

**Definition 2.11** An SSA edge for a variable $x$ corresponds to an edge pair $(e_1, e_2)$ satisfying the following properties:

1. there exists a definition of $x$ that reaches $e_1$,
2. there exists a use of $x$ reachable from $e_2$,
3. there is no assignment to $x$ on any control flow path from $e_1$ to $e_2$,
4. $e_1$ dominates $e_2$. 
Propagating information on an SSA graph guarantees that information is only propagated to points of interest: i.e., where the variables are defined or where lexically distinguished definitions of the same variable merge.

At the same time, limited control flow information is preserved for backward data flow analysis in SSA. We only keep track of occurrences of join nodes, which are where two definitions merge in forward analysis, while ignoring the occurrences of control branches. Those control branch nodes form the join nodes in an upside-down CFG. Since SSA form does not carry these nodes, we are unable to summarize information at control branches. As shown in Figure 2.1 when we propagate information backward along the SSA graph, we know that there are two definitions reaching the join node labeled $\phi$, but we have no knowledge of the control branch that decides which one of $\text{Def1}$ or $\text{Def2}$ is to be executed.

### 2.2.4 Data Flow Graph

The Data Flow Graph (DFG) representation was defined by Johnson and Pingali [8]. If we view SSA as factored Def-Use chains with extra nodes keeping track of merging points, we may view the DFG representation in a similar way. DFG uses switch operations to keep track of control branches and uses merge operations to keep track of join nodes.
DFG edges are defined in a similar way as SSA edges, but with more constraints:

**Definition 2.12** A DFG edge for a variable $x$ corresponds to an edge pair $(e_1, e_2)$ satisfying the following properties:

1: there exists a definition of $x$ that reaches $e_1$,
2: there exists a use of $x$ reachable from $e_2$,
3: there is no assignment to $x$ on any control flow path from $e_1$ to $e_2$,
4: $e_1$ dominates $e_2$,
5: $e_2$ post-dominates $e_1$, and
6: every cycle containing $e_1$ also contains $e_2$ and vice versa.

Conditions 5 and 6 guarantee that only SESE regions are bypassed by DFG edges. In Figure 2.2, we show the DFG representation of the same program shown earlier in Figure 2.1. DFG and SSA representations are the same for variable $x$. They both recognize the fact that the if-then construct forms an SESE region, and it does not contain interesting information relating to variable $x$. However, for variable $y$, where $y$ is defined within the if-then construct, DFG preserves more information than SSA. Using DFG, we can recover the control branch that decides which definition reaches the merge node.
2.3 Problem Formulation and Optimization Constraints

To improve the performance of TLDS execution model, the sources of the frequently occurring inter-thread data dependences are explicitly forwarded between threads to avoid data dependence violation. This process of forwarding creates a synchronization among the parallel executing threads. This synchronization limits the amount parallel execution possible.

We desire an instruction ordering such that all sources of thread-carried dependences in an earlier thread are computed and forwarded before their corresponding sinks in the later threads are executed. In this case, thread-carried data dependences create the need to move certain instructions earlier in the execution of a thread, while the thread-independent data dependences and control dependences become the constraints that prevent such code motion.

We first prioritize the forwarding paths that need optimization by their costs. Cost is defined as the product of the forwarding path length and the frequency that the forwarding should occur as estimated from profile information. These forwarding paths are optimized in the descending order of their costs. Scheduling the source of a thread-carried data dependence involves searching for all the computations on which it depends on and moving all computations earlier in the thread. This backward code motion is terminated when it is no longer possible to move the instructions further back or when the forwarding path is already smaller than the critical path between the threads.

In the rest of this section, we discuss the two constraints that prevent us from moving the source of a dependence in a thread: control dependence and data dependence.

2.3.1 Control Dependence Constraints

Dealing with Loops

Loops often need special consideration in data flow analysis. Loops that exist within the region that we intend to parallelize are usually loops with small loop counts and/or irregular memory access patterns, since these characteristics make them impractical to parallelize.
From the point of view of forwarding data, we are only interested in the final occurrence that produce the value to be forwarded to the next thread among all the write operations that modify the same data object in a thread. Similarly, among all the read operations in a thread, we are only interested in the first occurrence, when the value must be fetched from the previous thread. Therefore, for writes in an inner loop within a parallel region, the only interesting assignment occurs in the final iteration; similarly for loads within an inner loop of a parallel region, the significant occurrence is in the first iteration.

When the computation inside a loop does not contribute to the computation of the source of a dependence, we should treat the loop as an SESE region, and should not propagate information through the region. As we move instructions backward in the control flow graph, we should not attempt to move instructions into a loop, since that will increase the execution time of the thread. Moving instructions outside of a loop implies loop fission that divides the loop into two separate parts.

**No-Write Paths**

It is possible to have multiple writes modifying the same data object on a single execution path of a thread; it is also possible to have execution paths that do not have any modification for a particular data object. These paths are referred to as *no-write paths*. On these *no-write paths*, the value to be forwarded is computed in the previous iteration. To be more precise: when scheduling instructions for thread \( i \), we schedule forwarding on no-write paths to forward the value of a data object that is calculated in iteration \( i - 1 \) to iteration \( i + 1 \). To integrate the scheduling problem on no-write paths into the rest of scheduling algorithm, we augment the original CFG with a read at the beginning and a write at the end of the region, and construct the DFG for the variable of concern from the resulting CFG. No-write paths are identified as paths between the augmented read and write that have only switch and merge nodes. Two such paths are labeled in Figure 2.3. After the original writes are deleted from the DFG, all paths left in the graph that can reach END from START are no-write paths.

When a thread is executed, all control dependences along a no-write path must be resolved before it is known that the execution is following a no-write path. The last switch node on a no-write path contains the control branch that determines if the execution
follows a no-write path. Node 1 and 2 in Figure 2.3 are the last switch nodes on two no-write paths respectively. A special write is created and inserted at out-going edges of both the switch nodes that reach the augmented write directly. This special write is translated into a value forwarding instruction, to forward the value that is computed by the previous iteration. The case of no-write paths can now be integrated into the rest of the algorithm.

**Scheduling Across Control Branches**

Before an instruction makes its result effective, all control dependences that the instruction depends on must be resolved. Considering a region without data dependences, the value produced by a write operation is known as soon as all of its control dependences are resolved. The earliest points in the CFG indicating a write are the edges upon which the write is control dependent. Assuming no data dependences, we can move the write instruction to these edges, while still preserving the semantics of the program. Semantics are preserved in the sense that after the code motion, exactly the same instructions are executed on every path from START to END, and the same results are produced.

Let \( \{(u_1, v_1), (u_2, v_2), ..., (u_n, v_n)\} \) be the control dependent edges of instruction \( s \). If \( s \) is deleted and \( s' \) is inserted on all control dependent edges of \( s \), then any path from
START to END executes exactly the same codes after the code motion as before the code motion. No assumption is made regarding back edges in the above statement; hence the statement holds even in the presence of back edges. We can also show that for any instruction within a loop, one of its control dependent edges must be the back edge that defines the loop, while the others are entrances to the loop.

When we move a write operation around to a control branch, we can either schedule it speculatively across the control branches or terminate the code motion. Based on the execution frequency, there are three possibilities:

1: If the control flow favors the side of the branch containing the write (as in Figure 2.4), we can move the write above the control branch, and insert recovery code on the other edge.

2: If the control flow favors the side of the branch that does not contain the write, then we will stop moving the write instruction further back.

3: If the control flow is almost equally likely to go either way at the control branch (as in Figure 2.5), we can move the write instruction backward provided that we also move back the control branch.
2.3.2 Data Dependence Constraints

Thread-carried true data dependences create the need to schedule the sources and sinks of these dependences, and intra-thread dependences become constraints in scheduling.

Thread-independent data dependences impose constraints on how early a write operation can be evaluated, i.e., all operands of the write must be known. A write operation can be moved earlier in execution as long as all of its operands can be moved along.

Let $w$ be a write operation and $r$ be a read operation. In thread $i$, $w$ operation writes a value which is loaded in a later thread $i + 1$ by the $r$ operation. This creates a thread-carried RAW data dependence between the two, hence we would like to have the thread $i$ forward the value to thread $i + 1$ as early as possible. However this is not always possible. For example, when a loop is parallelized, all threads are executing the same code; and $r$ operation must be performed in thread $i$. The computation of $w$ in thread $i$ may depend on the result of the $r$ operation in the same thread. This creates a circular dependence, and poses an upper bound on how close we can schedule the read-write pair. We may recognize this constraint by creating a DFG that includes all data dependences and identifying strongly connected components in the graph.

Observation 2.1 If we identify a strongly connected component (SCC) of size greater than one in the data dependence graph, then we have encountered a circular dependence.

Observation 2.2 The number of computations on the circular dependence chain is equal to the size of the strongly connected component found in the DFG. In other words, the strongly connected component that is found in the DFG always forms a cycle.

Observation 2.3 In the optimal statement ordering, the forwarding path length between write and read is:

$$
\text{sizeof}(\text{SCC}) - 1
$$

where $\text{sizeof}(\text{SCC})$ is the largest strongly connect component that contains both the read and the write.

In the example illustrated in Figure 2.6, there exists a strongly connected component—including nodes {2, 3, 6}—of size 3. Therefore, the closest we can schedule the load and store pair is two.
2.4 The Complete Algorithm

Before we start to describe our scheduling algorithm, we first describe the data flow analysis problem that is used in our algorithm to compute useful information such as MayReach. Data flow analysis is described by three components [1]:

1: Domain of values: V;
2: Meet operator \((V \times V \rightarrow V)\), initial value;
3: A set of transfer function \(F: (V \rightarrow V)\).

Since all data flow analysis are described in a similar fashion, they can be treated with a unified framework. Such approach is described in [9].

2.4.1 Initialization

- **Loops:** Loops increase the complexity of our algorithm, since moving instructions across loop boundaries has a completely different meaning from moving them across forward edges. Loops are identified by back edges using the algorithms readily found in the compiler literature [1].

- **Approximated Path Length:** Approximated path length for a basic block is the average number of instructions executed from the beginning of the thread to the basic block, weighted by the execution frequency of the path. The path length that we compute provides an approximation on the length of the forwarding path be-
tween pairs of dependent writes and reads. This approximation is used to prioritize the order in which these forwarding paths are optimized.

Assuming no back edge, the path length can be approximated with simple data flow analysis by attaching an execution frequency to each branches. The analysis only iterates through the control flow graph once for this analysis. The transfer functions used are defined as follows:

\[
prob(bb) = \begin{cases} 
1 & \text{if } bb = \text{start} \\
\sum_{i \in \text{pred}(bb)} prob(i) \times PathProb(i, bb) & \text{otherwise}
\end{cases} \tag{2.3}
\]

\[
PathLength(bb) = \begin{cases} 
0 & \text{if } bb = \text{start} \\
\sum_{i \in \text{pred}(bb)} PathLength(i) \times prob(i) & \text{otherwise}
\end{cases} \tag{2.4}
\]

In the above function: \(\text{pred}(bb)\) is the set of predecessors of node \(bb\); \(PathProb(u, v)\) is the probability of the control flow exiting \(u\) take the edge \(u \rightarrow v\). \(Prob(bb)\), is the probability that a basic block \(bb\) is executed in a given thread.

We use a recursive algorithm to evaluate regions with inner loops. The algorithm starts with the innermost loop, and moves towards the outer loops. After the evaluation of an outer loop, we re-visit all components of the outer loop, and update the path length for each component. The algorithm in Figure 2.7 computes \(PathLength\) for every basic block in a given region, and stores the result in the floating point array \(\text{PATHLENGTH}\).

- \textbf{MayReach} is computed for all outgoing edges of all control branches. A write \(s\) belongs to the set \(\text{MayReach}(e)\), where \(e = (u, v)\), if there is a path \(\mathcal{P}\) from \(u\) to \(\text{END}\) and \(s \in \mathcal{P}\). The converse of \(\text{MayReach}(e)\) is the set of control branches that may reach \(s\) from \(\text{START}\). \text{MayReach} is formulated as a data flow analysis:

Domain: set of write operations
Lattice: \(\top = \text{a write } s \text{ is on some path to the END node}\)
\(\bot = \text{a write } s \text{ is not on any path to the END node}\)
Transfer: \(out = in \cup \text{current}\)
Meet: \(\cup_{e \in \text{pred}} out(e)\)
Initialization: NULL
/* This algorithm computes the path length of each basic block of a region. */
proc length(region, UPDATE)

/* Evaluate the inner-most loop first */
for each top-level loop l in the region
    length(l, TRUE)
end for

/* After evaluated all inner loops, compute path length for every basic block
   treating inner loops as single nodes */
Perform data flow analysis using the transfer equation Equation 2.3 and 2.4,
fill in the array PATHLENGTH[bb], for all bb ∈ region
/* Update all components of the region, if the region is a inner loop*/
sizeof(region) = PATHLENGTH[exit(region)];
RunLength(region) = AverageIterationCount(region);
if (UPDATE = TRUE)
    for each Basic Block bb
        PATHLENGTH[bb] = PATHLENGTH[bb] + RunLength(region) * sizeof(region);
    end for
end if
end proc

Figure 2.7: Algorithm for computing path length of each basic block of a region.
Cd and Cond were defined in Def. 2.5 and Def. 2.6, respectively. Pingali and Bilardi [15] described an efficient algorithm to compute the cd set for each node and cond set for each edge by solving the Roman Chariots Problem. The complexity of the algorithm is given by the following: There is a data structure APT for the problem \( T = \langle V, F, ROME, A \rangle \), which can be constructed in time

\[
T = O(|A| + (1 + 1/\alpha)|V|),
\]

and stored in space

\[
S = O(|A| + (1 + 1/\alpha)|V|),
\]

where \( \alpha > 0 \) is a design parameter. The complexity to obtain the cd is proportional to the output size, to obtain the \( \text{conds}(w) \) is in time \( O((1 + \alpha)s) \) where \( s \) is the output size.

- **Related Write-Read Sets:** Scheduling decisions for reads and writes that modify the same data object may be related. Therefore, it is useful to identify related reads and writes, and group them into sets, and make scheduling decisions for a set of operations. Data dependence profiling information provides a list of dependent pairs and the number of times the dependence occurs. Constructing a graph using the reads and writes as vertices, and putting an edge between those read and write that have thread-carried dependences, results in a bipartite graph. A typical profiling bipartite graph resembles what is shown in Figure 2.8, where several reads and writes related to the same variables are connected to form a set. We use a heuristic algorithm, as shown in Figure 2.9, to identify these groups, by incrementally building the groups starting from the edge with the highest number of occurrences.

### 2.4.2 Scheduling Algorithm

As we have discussed in Section 2.3.1, loops should be treated as combined nodes. In the algorithm in Figure 2.10, we assume all inner loops of the region are combined into one single node. Hence, there is no back edge.

Induction variables create recurrences that commonly generate thread-carried dependences. Before we apply the algorithm, we eliminate induction variables as described
We also eliminate any dependences that can be optimized away using loop shifting, as described in Section 1.2.

We make scheduling decisions for a set of related writes (as described in the previous section) at the same time. Each control branch has several outgoing edges, and each edge may reach one or more writes in the set. We associate an array called "marked" (of size equal to the total number of writes in the set that we are scheduling) with every outgoing edge of every control branches. If marked[store] = TRUE at a certain control branch then it indicates that we have already considered whether to move that write past that control branch. Among all the writes that may be reached from a control branches, we will decide to move one or more or none of them speculatively passing that control branch. We also associate an array called "up" (of size equal to the number of control branches) with each outgoing edge of every control branches to keep track of which write has been moved past that control branch.

The moveup() function, shown in Figure 2.11 is called by the overall algorithm to verify whether moving a write to its control dependence edges violates any data dependence. It takes three inputs: (i) the write operation to be moved; (ii) a list of instruction CHAIN that includes all instructions that must be scheduled with the write; and (iii) the destination to which the write is to be moved. This function returns a boolean decision that says whether the write can be moved to the destination. If so, CHAIN is the list of
/* This algorithm collects related reads and writes to form a set. */

LinkList *RelatedSets; /* List of related sets */

const int threshold; /* Dependence occur less than the threshold will not be collected */

int DependenceInfo[n][3]; /* Array of n data dependences as input. */

DependenceInfo[*][0] are the sources of the dependences,
DependenceInfo[*][1] are the sinks of the dependences,
dependenceInfo[*][2] are the number of times the frequency occur */

sort(DependenceInfo[*][2]);

/* sort DependenceInfo in ascend order of number of times the dependences occur. */

for (i = 0; i < n and DependenceInfo[i][2] > threshold; i++)

    RelatedSetsEntry HasRead = lookup(RelatedSets, DependenceInfo[i][0]);
    /* Search for the set that contains the read */

    RelatedSetsEntry HasWrite = lookup(RelatedSets, DependenceInfo[i][1]);
    /* Search for the set that contains the write */

    if (HasRead = nil and HasWrite = nil )
        /* If either read or right already belong to a existing set, create one. */
        NewSet = new RelatedSetsEntry(DependenceInfo[i][0], DependenceInfo[i][1]);
        append(RelatedSets, NewSet);
    end if

    if (HasRead = nil and HasWrite != nil )
        /* If write belongs to an existing set, add a new edge to the set. */
        AddEdge(HasWrite, DependenceInfo[i][0], DependenceInfo[i][1]);
    end if

    if (HasRead != nil and HasWrite = nil )
        /* If read belongs to an existing set, add a new edge to the set. */
        AddEdge(HasRead, DependenceInfo[i][0], DependenceInfo[i][1]);
    end if

    if (HasRead != nil and HasWrite != nil and HasRead != HasWrite)
        /* If the read and write belong to different sets, combine the sets */
        combine(HasRead, HasWrite);
    end if

end for

Figure 2.9: Algorithm for collecting related reads and writes to form sets.
/* This algorithm moves writes recursive to its control branches in the control flow graph. */
for (i = 0; i < NumOfSets; i++)
    while (MoreStore(set[i])) /*While there are still writes left to schedule */
        /* identify the most cost effective write to schedule */
        write = MostCostEffective(set[i]);
        ListOfInstruction CHAIN; /*This is the list of instructions to be rescheduled */
        append(CHAIN, write); /*Add write to the rescheduling list */
        next = conds(write); /*This is the control dependent edges of the write */
    while (next) /*Recursively move the writes to its control dependent edges */
        for (all e = (u, v) ∈ conds(st))
            for (each operand of write) /*Does it violate any data dependences*/
                move = move ∨ moveup(operand, CHAIN, e); 
            end for
        if (move = false) /*The write cannot be moved back due to data dependence*/
            tail = tail(CHAIN);
            for each element in CHAIN
                delete the original computation;
                append the computation at tail;
            end for
        end for
        for all control branches
            if (mayreach(store)) mark(store) = true; end if
        end for
        break; /* Code motion failed, stop code motion */
    end if
if (!on_most_frequently_executed_path(e, head(e)) break ; /*e is not on the most frequented path, stop code motion */
else up(store) = true;
    next = cond(next); /*Code motion succeeded, move it further back */
end for
end while
end while
end for

Figure 2.10: The overall code motion algorithm.
instructions that have to be moved along.

2.5 Examples

This study targets non-numeric applications with ambiguous memory accesses which are hard if not impossible to analyze at compile time. Our algorithm attempts to identify the sources of thread-carried dependences that are scattered over the region that we intend to parallelize, and schedule them early in the thread. Part of the challenge to our algorithm involves scheduling instructions across a large number of instructions, including control branches. We use several examples to illustrate different aspects of our algorithm. These examples are from the SPEC92 and SPEC95 benchmark suites.

2.5.1 Regions Without Inner Loops

The first example uses a loop from go as a potentially parallelizable region, each iteration of the region becomes a thread. There are only forwarding edges in each thread. The region consists of four levels of if-branches. Depending on the outcome of the branch, a different value is assigned to the variable “state”. The value of “state” from an earlier thread becomes the control branch condition on the top-level if-branch of the later thread. There is clearly a true data dependence between two consecutive threads. The variable involved is not an induction variable, hence cannot be eliminated as such. Also, the dependence cannot be eliminated with loop shifting algorithm, since the data dependences form a cycle. Hence, we try to apply our algorithm to schedule the instructions, so that the value of “state” from an earlier thread is available to the later thread as early as possible.

The original program is shown in Figure 2.12, the control flow graph is shown in Figure 2.13, where all edges are labelled with numbers that denote how many times that particular branch is taken.

The CFG after code motion is shown in Figure 2.14, where nodes 6, 7 and 8 are speculatively executed before node 5, although these nodes are control dependent on node 5. We also identify no-write paths in our algorithm, and insert forwardings at all paths where no modification to the variable occur. The program after code motion is shown in Figure 2.15, the diamond symbols (◊) mark all forwarding instructions.
/* Function moveup checks if the code motion violates any data dependence. */

int moveup(st, CHAIN, TOP)
    for each operand (op) of st
        construct_data_flow_graph(TOP, op);
        if (moveup(operand, CHAIN, TOP) = FALSE) then return FALSE;
        else add operand to CHAIN;
        if incoming = TOP return TRUE;
        if incoming = NULL return TRUE;
        if there is still no need to move up return FALSE;
        if (pred.type = switch) /*Incoming node is a control branch */
            /*Speculatively schedule the write ahead of the control branch */
            if (num of times(pred)/total) > threshold
                move it up speculatively, leave another side speculative;
                place CHAIN on the other side;
                return moveup(control, dependence, TOP);
            /*Move the entire control branch */
            else if (num of times(pred)/total) > threshold2)
                append(pred, CHAIN);
            end if
        end if
    end if
    /*Keep moving the instruction back in all other cases */
    if (pred.type = join)
        for each incoming edge
            moveup(op, CHAIN, TOP);
        end for
    end if
    end if
    if (pred.type = write) Perform variable renaming; end if
    if (pred.type = load) return FALSE;
end for
end proc

Figure 2.11: Function moveup checks if the code motion violates any data dependence.
for (i = 0; i < boardsize-4; ++i) {
    if (state == 0) {
        if (board[s1] != NOGROUP || board[s2] != NOGROUP ||
            state = 1;
    }
    else if (state == 1) {
        if (board[s1] == NOGROUP board[s2] == NOGROUP
            state = 2;
        if (edgeshimari(s3-longincr,longincr,shortincr))
            ++shimari;
        startext = s1 - longincr;
    }
    else if (state == 2) {
        if (board[s1] != NOGROUP || board[s2] != NOGROUP ||
            state = 1;
            if (edgeshimari (s3,-longincr,shortincr))
                ++shimari;
            endext = s1;
            findbestextension(startext,endext,longincr,shortincr,shimari);
            shimari = 0;
        }
    }
    s1 += longincr; s2 += longincr; s3 += longincr;
    s4 += longincr; s5 += longincr;
}

Figure 2.12: A loop from go: a region with four levels of if-branches.
a.) The CFG. All edges are labeled with number of times the edge is taken.

b.) The DFG.

Figure 2.13: The CFG and the DFG of the region with four levels of if-branches.
Figure 2.14: The CFG of the region with four levels of if-branches after instruction scheduling
/* A region with multiple levels of if-branches after code scheduling */
for (i = 0; i < boardsize-4; ++i) {
    if (state == 0) {
        ◊ forward (state = 1);
        if (board[s1] != NOGROUP || board[s2] != NOGROUP ||
        ◊ else forward (state);
    }
    else {
        if (temp = board[s1] == NOGROUP && board[s2] == NOGROUP &&
        ◊ forward (state = 2);
        if (state == 1) {
            if (temp)
                if (edgeshimari(s3-longincr, longincr, shortincr)) ++shimari;
                startext = s1 - longincr;
        }
        ◊ else forward (state); }
    else if (state == 2) {
        if (board[s1] != NOGROUP || board[s2] != NOGROUP ||
            /* Kill next thread, because wrong value has been forwarded */
            ◊ forward (state = 1);
            if (edgeshimari(s3,-longincr, shortincr)) ++shimari;
            endext = s1;
            findbestextension(startext,endext,longincr,shortincr,shimari);
            shimari = 0; }
    }
}

Figure 2.15: A region with four levels of if-branches after code scheduling. (◊ marks the forwarding instruction)
2.5.2 Regions With Inner-Loops

We use the main loop from compress to demonstrate how inner loops are handled. Over 99% of the execution time of compress is spent on the main loop, the loop steps through each character from the input file. The code is shown in Figure 2.17, the CFG of the region is shown in Figure 2.16, the dotted line is the back edge that defines the inner loop. The variable that creates the longest dependence chain is ent. There are three exits that lead to the termination of the epoch, all exits are dominated by one of the three assignments of ent, and there is no no-write path. The dependence pair that has the highest cost in performance is (write3, read1), hence the forwarding path between write3 and read1 should be optimized. We attempt to schedule write3 early in the region by moving write3 to both of its control dependence edges (2 → 3) and (10 → 12). This part of the code motion is trivial, because the data dependence graph between (2 → 3), (10 → 12) and write3 is simple—there is no data dependence. The resulting control flow graph is shown in Figure 2.18, with two copies of write3 on edge (2 → 3) and edge (10 → 12) respectively. We could not move either of the two copies of the writes further back to the control dependence edges of node 2 and 10, because both edges (2 → 3) and (10 → 12) are not the frequently executed paths and should not be optimized.

2.5.3 Loop Splitting

In the discussion regarding inner loops in Section 2.3.1, we concluded that moving instructions out of an inner loop results in loop fission. Loop fission requires we create a new loop for those instructions that we intend to move out of the inner loop, and reproduce the loop indexing information for it.

We show a region that needs loop fission from espresso in SPEC92. The CFG and DFG of the loop are shown in Figure 2.20. The back edge of the loop that needs to be split is shown with dotted line. The DFG, shown in Figure 2.20b, indicates that the computation of a[0] and a[i] could be moved up against the direction of control flow along with the loop that contains it. Figure 2.20c shows part of the loop being speculatively executed in the beginning of the region, while the second part not being moved. The code after code scheduling is shown in Figure 2.21.
Figure 2.16: The main loop in compress contains a back edge.
/* Main loop of compress: A region with back edge */
while ( (c = getchar()) != EOF ) {
    incount++;
    fcode = (long) (((long) c << maxbits) + ent);
    i = ((c << hshift) xor ent);
    if ( htabof (i) == fcode ) {
        ent = codetabof (i); continue ;
    } else if ( (long)htabof (i) < 0 ) goto nomatch;
    disp = hsizereg - i;
    if ( i == 0 ) disp = 1;
    probe: if ( (i -= disp) < 0 ) i += hsizereg;
    if ( htabof (i) == fcode ) { ent = codetabof (i); continue; }
    if ( (long)htabof (i) > 0 ) goto probe;
    nomatch: output ( (codeint) ent );
    outcount++; ent = c;
    if ( freeent < maxmaxcode ) {
        codetabof (i) = freeent++;
        htabof (i) = fcode;
    } else if ( (countint)incount >= checkpoint blockcompress ) clblock ();
}

output( code ) codeint code; {
    bp += (roff >> 3); bp++;
    if ( bits >= 8 ) { *bp++ = code; code >>= 8; bits -= 8; }
    if (bits) *bp = code;
    offset += nbits;
    if ( offset == (nbits << 3) ) {
        bp = buf;
        bits = nbits;
        bytesout += bits;
        do putchar(*bp++); while (--bits); offset = 0;
    }
}

Figure 2.17: The CFG and DFG of the main loop in compress, it contains a back edge.
Figure 2.18: The main loop of compress after instruction scheduling
262 for (; (a = *A1++) != NULL; ) {
263   if (a[0]  ACTIVE) {
266     (void) setmerge(lift, bcube, a, mask);
134     { /* Procedure in-lined */
135       register int i = (bcube[0]  0x03ff);
136       lift[0] = 0x03ff, lift[0] |= (i)
136     do lift[i] = (a[i]mask[i]) | (a[i] mask[i]); while (--i > 0);
137       return r;
138     }
269   for (B2 = B1; (b = *B2++) != NULL; ) {
271     if (! setpimplies(a,b)) continue ;
207     { /* Procedure in-lined */
208       register int i = (a[0]  0x03ff);
209     do if (a[i]  b[i]) return FALSE; while (--i > 0);
210       return TRUE;
211     }
274     setor(a, a, liftor);
103     { /* Procedure in-lined */
104       register int i = (a[0]  0x03ff);
105       a[0] = 0x03ff, a[0] |= (i)
105     do a[i] = a[i] | liftor[i]; while (--i > 0);
106       return a;
107     }
275     break ;
276   }
277 }
278 }

Figure 2.19: A loop from espresso, loop fission is needed to move part of the loop back.
The back edge of the loop that requires loop fission

(a) Control flow graph
(b) Data flow graph
(c) Control flow graph after code motion

Figure 2.20: Loop scheduling that requires loop fission
262 for (; (a = *A1++) != NULL; ) {
269   for (B2 = B1; (b = *B2++) != NULL; ) {
274     setor(a, a, liftor);
278     
103     { register int i = (a[0] 0x03ff);
105     a[0] = 0x03ff, a[0] |= (i);
109     forward(a[0]);
110     do a[i] = a[i] { liftor[i];
111     forward(a[i]);
113     while (--i > 0);
115     return a;
278   }
279 }
280 if (a[0] ACTIVE) {
283   (void) setmerge(lift, bcube, a, mask);
288   { register int i = (bcube[0] 0x03ff);
291   lift[0] = 0x03ff, lift[0] |= (i)
293   do lift[i] = (a[i]mask[i]) | (a[i] mask[i]); while (--i > 0);
295   return r;
296 }
299 for (B2 = B1; (b = *B2++) != NULL; ) {
302   if (! setpimplies(a,b)) continue;
307   { register int i = (a[0] 0x03ff);
309   do if (a[i] b[i]) return FALSE; while (--i > 0);
313   return TRUE;
314 }
315 break ;
316 }
317 }
318 }
319 }
320
Figure 2.21: A loop from espresso, the loop is split into two parts. Part of the loop is moved to the beginning of the region. (◊ marks the forwarding instruction)
Chapter 3

Algorithm Evaluation

Having described our instruction scheduling algorithm for reducing the critical forwarding path length in the previous chapter, we now estimate the effectiveness of this algorithm. Algorithm effectiveness is evaluated by comparing the average number of non-overlapped instructions in parallel execution before and after the optimization. These regions, presented as examples in this chapter, are found in applications from the SPEC95 and SPEC92 integer benchmark suites; they generally account for a significant fraction of the total execution time of the applications to which they belong. In the rest of this chapter, we first discuss the implementation involved and explain how this optimization pass fits into the compiler framework. We then discuss the method that we use to evaluate the algorithm. Finally we present and discuss the performance improvement potential of this algorithm.

3.1 Implementation Framework

The implementation of this thesis is done within the SUIF (Stanford University Intermediate Format) compiler system. The SUIF compiler system is designed as a flexible framework for compiler optimization research. It allows multiple optimization passes to be inserted easily between the front-end and the code generation pass. It also provides annotations that can be used to attach information to almost all data structures in the SUIF representation. For our purposes, annotations are useful for transferring profile information and forwarding decisions across several compiler passes.

Figure 3.1 illustrates the compiler framework used in this thesis. We have imple-
mented the control flow profiling pass to collect control flow information. These data are used to compute statistical information that describes branch and procedure call behavior. These information are then attached to the original program as annotations. We then identify regions that are worthy of being parallelized and collect data dependence information in these regions. Using data dependence information, we identify data objects that need to be forwarded between threads. The next step in compilation is to perform code scheduling and to specify forwarding. However, this pass has not yet been implemented. Finally, we perform register allocation and generate assembly code.

Profiling passes are implemented in three stages: We first create a library containing a collection of profiling routines that, when called, perform counting and bookkeeping operations. Procedure calls to these routines are inserted into the original program at all points of interest. The program is then recompiled and linked with the profiling routines. The routine _startup_prof initializes the hash table, while _cleanup_prof formats the content of the hash table and prints it to a file. The rest of the routines record information by updating hash table entries. The file generated by _cleanup_prof is used in the third stage of our profiling pass. In this stage the raw data are analyzed and attached to the SUIF representation of the program as annotations.

3.2 Evaluation Methodology

In this section, we verify the effectiveness of our algorithm and demonstrate the potential performance improvement we may obtain with instruction scheduling on real applications. Performance is normally measured in terms of application execution time. In this case, we were unable to simulate the execution of the applications nor were we able to obtain execution time; instead, average instruction counts for non-overlapped instructions of these regions are used to describe the performance. Performance improvement of an optimization pass is evaluated as the ratio of the average instruction counts before and after the optimization, normally referred to as speedup. Before we present a first order approximation of the performance improvement of our algorithm, we first describe the evaluation process.
Figure 3.1: Compiler Framework
3.2.1 Performance Estimation

The efficiency of parallel execution is determined by the non-overlapped portion of the program. In the presence of explicit forwarding, the non-overlapped portion of execution is determined by the critical forwarding path length. In this evaluation, forwarding path length is approximated as the difference between the path lengths of the source and the sink of the dependence. Forwarding path length is calculated as:

\[
\text{forwarding path length} = \begin{cases} 
0 & \text{if length(sink) > length(source)} \\
\frac{\text{length(source)} - \text{length(sink)}}{\text{distance(source, sink)}} & \text{otherwise}
\end{cases}
\]  

(3.1)

The distance between the source and the sink of a dependence is equal to the iteration number of sink subtracting the iteration number of the source of the dependence.

How much will this forwarding delay degrade the performance of the microprocessors? First we define the following notations:

- \( P \) = the number of processors,
- \( N \) = the total number of threads to be executed,
- \( l \) = the length of a single thread in terms of number of SUIF instructions,
- \( d \) = the length of the critical forwarding path length between two consecutive threads in terms of number of SUIF instructions.

Intuitively, when the forwarding path length is smaller than \( \frac{l}{P} \), the execution start-time of the \((N + 1)th\) thread depends on whether there are sufficient resources (processors). However, when the critical forwarding path length is greater than \( \frac{l}{P} \), the execution start-time of the \((N + 1)th\) thread does not depend on the availability of the processor alone; instead, it depends on how early the data dependences are resolved. In the latter case, parallelization is not possible due to the sequential characteristics of the application. Figure 3.2 presents both situations, and offers two different ways of estimating the instruction counts. Figure 3.2(a) shows the case where parallelism is limited by resources, and the instruction count is given by:

\[
\text{instruction count} = \frac{(l \times N)}{P}
\]  

(3.2)
a.) Forwarding path length = 0

b.) Forwarding path length is small

c.) Forwarding path length is large enough to determine execution time

Figure 3.2: Execution time estimation with different forwarding path lengths
When a forwarding path causes delay, two possibilities exist: when the critical forwarding path length is small, it does not dominate the execution time of the region, as shown in Figure 3.2(b), the instruction count is given by:

$$\text{instruction count} = (l \times N) + P + (P - 1) \times d$$ \hspace{1cm} (3.3)

If \( N \gg P \), the second term is negligible, the equation is reduced to:

$$\text{instruction count} \approx (l \times N) + P$$ \hspace{1cm} (3.4)

However when the critical forwarding path is long, that is, \( d \) is large (larger than \( \frac{1}{P} \)), the execution time is determined by \( d \). When \( d > \frac{1}{P} \), the execution time becomes a function of the critical forwarding path length and the number of threads to be executed, as shown in Figure 3.2(c), that is:

$$\text{instruction count} = d \times (N - 1) + l$$ \hspace{1cm} (3.5)

When \( N \gg 0 \), the equation is reduced to:

$$\text{instruction count} = d \times N$$ \hspace{1cm} (3.6)

### 3.2.2 Accuracy of Evaluation

The execution time is approximated as the number of instructions that cannot be parallelized. This approximation is calculated from the static program and profiling information. Although this approximation does not accurately reflect the execution time of the application, it is sufficient for the purpose of illustrating the amount of parallelism that we can extract with our instruction scheduling algorithm.

Code scheduling may change application performance in many ways. We will briefly discuss some of the important factors here.

#### Register Pressure

As we move the sources of data dependencies earlier in the execution of a thread, the lifetimes of some data that live in the register change (we forward and schedule both memory data objects and scalars that live in the registers). As illustrated in Figure 3.3, the variable of interest is \( a \). The index indicates different definitions of the variable and
their corresponding uses. Originally the lifetimes of \( a_0 \) and \( a_1 \) do not overlap, and they share one register. After the code motion, the lifetimes of \( a_0 \) and \( a_1 \) overlap and they require two registers. Also, the lifetime of \( a_0 \) is much longer than it is before the motion, which creates a larger possibility to overlap with the lifetimes of other registers.

**Cost of Speculative Execution**

Code motion and speculative execution are not cost-free. Since we have allowed instructions to be scheduled speculatively to the control branches on which they depend, we have to provide mechanisms to squash the threads and start over again when they receive the wrong value. When this happens, performance is degraded. There is a trade-off between the frequency of recovery \( \times \) the cost of recovery and the performance gain obtained through code motion.

### 3.3 Results

Table 3.1 describes the regions under investigation. Information such as the applications to which the regions belong, the percentage of the total execution time the applications spend in these regions, and the sizes of the regions as the average number of SUIF instructions on each path weighted by path frequency is all included. *Critical Forwarding*
Table 3.1: Benchmark characteristics

<table>
<thead>
<tr>
<th>ID</th>
<th>Type of Region</th>
<th>Loop Count</th>
<th>Size of Region</th>
<th>Critical Forwarding Path</th>
<th>From</th>
<th>Percentage of Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Loop</td>
<td>13735</td>
<td>317</td>
<td>317</td>
<td>compress: compress.c:782</td>
<td>99.98%</td>
</tr>
<tr>
<td>2</td>
<td>Loop</td>
<td>63</td>
<td>65</td>
<td>44</td>
<td>jpeg: jchuff.c:365</td>
<td>35.33%</td>
</tr>
<tr>
<td>3</td>
<td>Loop</td>
<td>13</td>
<td>253</td>
<td>230</td>
<td>espresso: complec:269</td>
<td>4.42%</td>
</tr>
<tr>
<td>4</td>
<td>Loop</td>
<td>83</td>
<td>1944</td>
<td>972</td>
<td>go: go25.c:772</td>
<td>13.20%</td>
</tr>
<tr>
<td>5</td>
<td>Loop</td>
<td>$4.5 \times 10^5$</td>
<td>1672</td>
<td>1672</td>
<td>m88ksim: go.c:118</td>
<td>99.13%</td>
</tr>
<tr>
<td>6</td>
<td>Loop</td>
<td>8.36</td>
<td>124</td>
<td>121</td>
<td>go: g25.c:829</td>
<td>3.31%</td>
</tr>
<tr>
<td>7</td>
<td>Loop</td>
<td>73.7</td>
<td>74241</td>
<td>0</td>
<td>sc: interp.c:994</td>
<td>40.49%</td>
</tr>
</tbody>
</table>

Path is the forwarding path with the largest cost, which is calculated as the product of the forwarding path length and the execution frequency of the forwarding path.

Table 3.2 demonstrates the algorithm effectiveness by comparing the instruction counts and the critical forwarding paths before and after the optimization. The instruction counts for parallel execution are calculated for the case of four processors. We assume that the time it takes for the processors to communicate a value is small enough, relative to the size of the threads, and can be ignored. We use average instruction count of the sequential execution as the base case to estimate the speedup of the parallel execution. Since we allow the compiler to schedule instructions speculatively on the most frequently executed path, we also estimate the potential performance degradation caused by speculation violations. The cost of speculation violations is estimated as the product of the thread length and the frequency of dependence violations; the latter is calculated from the profiling information.

Most of the regions studied achieve little or no speedup when executed on the TLDS architecture without compiler optimization. This is caused by the long forwarding paths. Only two regions achieve good speedup without optimization, and in both cases they have short critical forwarding paths. For all other regions, the critical forwarding path lengths are reduced significantly by the optimization. For three of the regions the critical
Table 3.2: Performance improvement through instruction scheduling

<table>
<thead>
<tr>
<th>ID</th>
<th>Execution Time</th>
<th>Before Optimization</th>
<th>After Optimization</th>
<th>Execution Time with Recovery Cost</th>
<th>Speedup With Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1277457</td>
<td>1236252</td>
<td>79.25</td>
<td>1</td>
<td>461085</td>
</tr>
<tr>
<td>2</td>
<td>4095</td>
<td>3795</td>
<td>44.53</td>
<td>1.07</td>
<td>1575</td>
</tr>
<tr>
<td>3</td>
<td>3289</td>
<td>2990</td>
<td>230</td>
<td>1.1</td>
<td>822</td>
</tr>
<tr>
<td>4</td>
<td>1602352</td>
<td>400538</td>
<td>972</td>
<td>4</td>
<td>400538</td>
</tr>
<tr>
<td>5</td>
<td>7.6x10^8</td>
<td>7.6x10^8</td>
<td>1176</td>
<td>1</td>
<td>1.9x10^8</td>
</tr>
<tr>
<td>6</td>
<td>1036.64</td>
<td>1011.56</td>
<td>121</td>
<td>1</td>
<td>384.56</td>
</tr>
<tr>
<td>7</td>
<td>742441</td>
<td>18560.25</td>
<td>0</td>
<td>4</td>
<td>18560.25</td>
</tr>
</tbody>
</table>

Path lengths are reduced by factors greater than four. After estimating the speedup on four processors, we observe that all parallelized regions perform 2.5-4 times better after optimization compared with the non-optimized cases on TLDS architecture. Speculation violations degrade performance slightly, yet performance improvement is still considerable. Figure 3.4 summarizes our calculations.
Figure 3.4: Potential regional speedups on four processors (P = No critical forwarding path minimization, S = Using the critical forwarding path minimization algorithm to perform code motion, R = Including the cost of recovery operation from failed speculation)
Chapter 4

Conclusions

In this thesis, we investigate the degree of parallelism that we can extract from non-numeric applications on a TLDS architecture with compiler controlled instruction scheduling. To fully exploit thread level parallelism and to achieve the full potential provided by the TLDS architecture, we have proposed an algorithm that schedules the sources of thread-carried data dependences as early as possible in the thread, so that their corresponding sinks in a later thread do not have to stall and wait for the values to become available. This work is prompted by the observation that in many regions where parallelization is desired, data dependences frequently occur between consecutive threads and synchronization is required between the producer and consumer threads. In this case, the execution time of the parallelized region is determined by the non-overlapped portions of the threads. The goal of this optimization is to minimize the critical forwarding path lengths between the sources and the sinks of the thread-carried data dependences. Our algorithm can significantly reduce the critical forwarding path lengths on some potentially parallelizable regions and increase the degree of parallelism that we can extract from those applications. Our algorithm performs the following:

- Identifying thread-carried dependent pairs and minimizing the length of the critical forwarding path between the source and the sink of the dependence by scheduling source as early as possible.

- Aggressively scheduling instructions on frequently executed paths by speculatively moving certain computations ahead of their control dependences, if doing so can reduce the forwarding path length on frequently executed paths.
We studied several parallelizable regions taken from SPEC92 and SPEC95 benchmark suites. By comparing the estimated performance improvements and critical forwarding path lengths of these regions before and after optimization, we concluded that our code scheduling algorithm can substantially improve performance of speculative executions on TLDs. Most of the regions that achieve little or no speedup on the TLDs architecture before the instruction scheduling achieve speedups of 2.5 to 4 after the instruction scheduling. Speculation violations and recovery operations degrade performance slightly, but all regions still enjoy considerable performance improvement with our optimization.

4.1 Future Directions

The next step in our study is to implement the instruction scheduling algorithm that is presented here and evaluate the performance benefits of this algorithm with a detailed simulator.

In chapter 3, we have described the methods that are used to estimate the performance benefits of our algorithm, these methods ignore the second order effects (such as register pressure, cache behavior, prefetching behavior) caused by code scheduling that may degrade performance. It is also important to develop techniques that can remedy such problems in future algorithms. One possible solution is to take into account both the parallelism extracted by a certain code motion and the performance degradation caused by these second order effects, and to decide if a certain code motion is desirable with a cost-benefit model.
Bibliography


in average data capsule delays when traffic set is all external and when traffic set is mixed or all local. This is because when traffic set is all external, the scheduler has unlimited access to virtual time information of arriving queues. Hence a data capsule is transmitted as long as there is no capsule with smaller virtual time tag currently in the system. When traffic set is all local or mixed, a data capsule cannot be transmitted immediately even if there is no capsule currently in the system, it must waiting for the tag polls (which has smaller virtual time tags than virtual time tags of data capsule i) to remote arriving queues. This incurs a significant delay for each data capsule. Note that the delay plots are not always increasing as offered load increases. This is because average delay is calculated only for capsules that are transmitted. In some cases, an increase in offered load would not cause a significant increase in delay since time-expired capsules have been discarded.
Figure 5.3: DFQ capsule loss
CHAPTER 5. SIMULATION RESULT ANALYSIS

The average delay for data traffic is expressed in logarithm scale in order to cover delay values at all input load.
Figure 5.5: DFQ capsule delay distribution

The mixed traffic set at offered load of 8 Mbps or 0.32 of system capacity.

Figure 5.5 illustrates the capsule delay distributions of various traffic types when mixed traffic set has offered load of 8 Mbps. This figure provides a insight to how capsules of different media and localities are served. Since all capsules are served according to a universal virtual clock, locality of a capsule does not affects the delay it experiences. We see that for a given medium type (may that be voice, video or data), the delay distributions are the same for local-to-local, local-to-external and external-to-local traffic. Video traffic in general is served first since it has the largest resource share allocation. It is possible to transmit a video capsule immediately after
its arrival. Therefore, most of video capsule delays are distributed under 5 ms. Voice capsules cannot be transmitted immediately after their arrivals, because the system needs to poll video queues. Therefore, voice traffic is served 5-6 ms after arrival at the earliest. The data services have the least resource shares allocated to them, therefore experience the worst delay. For each data capsules transmitted, there are numerous tag and data polls to video and voice queue. This in turn incurs a very large delay to data capsules. The fact that data capsules arrive in bursty pattern causes even larger delay and capsule loss probability.

**DFQ performance vs channel error probability**  
Figure 5.6 shows the overall system performance as a function of channel error probability. Since this channel error probability is equivalent to capsule error probability, we uses capsule error probability instead for clarity reasons. As the capsule error probability increases, the probability of a capsule experiencing larger delay also increases, which in turn results in an increase in the number of capsule being dropped. Since some capsules with large delay are removed from the system, the over-all delay decreases initially. However as capsule error continues to rise, the delay due to retransmission eventually has some influence on transmitted capsules. Hence the overall average capsule delay starts to rise.

When the error probability is reasonable (i.e. under 0.01), channel error has very little effect on system performance. (Note that capsule loss only rises from 0.266 to 0.275 on an already every heavily used channel, and the average delay is affected even less.) However, this is at the cost of adding an acknowledgment overhead, which is not necessary if the transmission channel is error free. As the capsule error rises to 0.1, the effect of channel error on capsule loss and average delay becomes very apparent. The capsule loss has increased 0.12 and average capsule delay has increase 10 ms.
Figure 5.6: DFQ performance in errored environment
The mixed traffic set at load of 12 Mbps or 0.48 of channel capacity.

Performance vs number of remote stations  Figure 5.7 shows the overall capsule delay and loss as number of stations increases. Because voice and data queues are normally non-empty at heavy load, tag polls generated for voice and data traffic is insignificant compared to tag polls generated for video traffic. Therefore in order to understand system performance, we must look at the tag poll generating pattern for video traffic as the number of remote stations in the system increases. When the number of stations is few, every remote station has multiple number of video connections. Hence tag polls generated for video traffic are few. As a result, capsule loss is small. An increase in remote stations means that video connections become
more spread out, therefore more tag polls to video traffic are generated and larger capsule loss ratio is resulted. As the number of remote stations continues to grow, eventually the number of remote stations becomes more than the number of video connections. At this point, tag polls generated for video connections no longer increases as the number of remote stations increases. Consequently, the capsule loss ratio becomes more or less flat. The slight rise in capsule loss is due to a slight increases in data and voice traffic tag polls as number of station increases.

![Graph](image)

**Figure 5.7:** DFQ performance vs number of stations

The mixed traffic set at load of 12 Mbps or 0.48 of channel capacity.
5.2 Modified DFQ Simulation

5.2.1 Modified DFQ Simulation Scenarios and Parameters

In order to better compare the performance of DFQ and modified DFQ. The simulation scenarios for modified DFQ remain the same as for DFQ simulation. Simulations are also performed to obtain DFQ performance versus number of remote stations and channel error probability.

Modified DFQ simulation parameters are given in Table 5.4. All transmission intervals have taken guard time and synchronization time into account. Note that transmission intervals are shorter than those in the DFQ system. This is because virtual time information no longer needs to be included. Substituting these parameters into (3.1), the maximum modified DFQ system throughput is obtained (Table 5.5). The modified DFQ maximum throughput is slightly better than those of the DFQ system. However, MAC overhead still occupies a significant portion of channel bandwidth.

<table>
<thead>
<tr>
<th>System Channel Bandwidth (C)</th>
<th>25Mbps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tx/Rx turnaround time (T_{turnaround})</td>
<td>2 μs</td>
</tr>
<tr>
<td>Capsule transmission interval (T_c)</td>
<td>21 μs</td>
</tr>
<tr>
<td>Data poll transmission interval (T_{data poll})</td>
<td>2 μs</td>
</tr>
<tr>
<td>Ack transmission interval (T_{ack})</td>
<td>1 μs</td>
</tr>
<tr>
<td>Data poll interval (T_{poll int})</td>
<td>5 ms</td>
</tr>
</tbody>
</table>

Table 5.4: Modified DFQ simulation parameters

<table>
<thead>
<tr>
<th>Traffic sets</th>
<th>MAC efficiency ($\eta_{mac}$)</th>
<th>Capsule efficiency ($\eta_{cap}$)</th>
<th>Max. throughput ($\eta_{mac} \times \eta_{cap}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>all external</td>
<td>0.807</td>
<td>0.690</td>
<td>0.557</td>
</tr>
<tr>
<td>all local</td>
<td>0.705</td>
<td>0.690</td>
<td>0.483</td>
</tr>
<tr>
<td>mixed</td>
<td>0.759</td>
<td>0.690</td>
<td>0.524</td>
</tr>
</tbody>
</table>

Table 5.5: Modified DFQ maximum throughput
5.2.2 Modified DFQ Simulation Results and Analysis

Modified DFQ general performance  Figure 5.8 depicts modified DFQ system throughput with different input traffic localities. Comparing with DFQ throughput performance, modified DFQ boosts the system maximum throughput from 0.28 to 0.48 for the all local traffic set, and from 0.35 to 0.52 for the mixed traffic set. All traffic sets have achieved the maximum system throughput as calculated in table 5.5. In addition, because modified DFQ eliminates any virtual time related overheads, throughput does not level off when incoming traffic becomes heavy.

Figure 5.8: Modified DFQ system throughput

Figure 5.9 and figure 5.10 are the average delay and loss comparisons of voice,
data, video traffic in different traffic sets. It is observed that for the all local traffic set (worst case in terms of system throughput), the system does not start to discard capsules until the offered load is at 0.45 of system capacity, which is 94% of the maximum system throughput. This is quite encouraging. Even though the maximum throughput due to MAC and capsule overheads is fairly low in this simulation, it is feasible in the future that with better hardware support, MAC and capsule overheads (especially the turn around time) can be reduced, and maximum system throughput will be raised. Of course, the maximum channel utilization level $u_{Q_{os}}$ at which guaranteed QoS is met also depends on offered traffic mixes. $u_{Q_{os}}$ will be lower if all incoming traffic is delay sensitive than if all traffic is non-delay sensitive.

The delay experience by a capsule consists of two parts: access delay and queueing delay. Access delay refers that when an empty remote queue $i$ is polled by the base station every $5\ ms$, any new arrivals at remote queue $i$ during this $5\ ms$ interval must wait until the next poll in order to let the base station know that there are capsule waiting in remote queue $i$. Queueing delay refers to, given that the base station acknowledges that a capsule waiting the system, this capsule must wait until all capsules with higher priority are transmitted. External-to-local traffic only has queueing delay; whereas local-to-local and local-to-external traffic has both access and queueing delay. At lower load, capsule delays are mainly access delay. This is illustrated by the flat delay curves for the all local and mixed traffic sets in figure 5.10. As the traffic load continue to increase, eventually the channel becomes congested. At this point, queueing delay becomes the dominating delay, and the average delay increases sharply as traffic load increases.
Figure 5.9: Modified DFQ capsule loss
Figure 5.10: Modified DFQ average capsule delay
Data delay are measured in logarithm scale in order to cover average data delays at all input loads.
CHAPTER 5. SIMULATION RESULT ANALYSIS

Figure 5.11: Modified DFQ capsule delay distribution

The mixed traffic set at offered load of 12 Mbps or 0.48 of channel capacity.

Figure 5.11 illustrates the capsule delay distribution of a mixed traffic set at load of 12 Mbps. Clearly since external-to-local traffic does not incur access delay, its capsule delays are generally distributed within the first 5 ms interval. Both local-to-external and local-to-local traffic has access delay and similar distribution curves. The flat parts of distribution curves are due to fixed 5 ms polls to empty remote queues. The data capsule delay distribution is more spread out because data capsules have the lowest priority. Hence, the larger the probability of a data capsule to experience a longer delay.
Modified DFQ performance vs channel error probability Consider modified DFQ performance in errored environment in figure 5.12, both delay and loss increase as capsule error probability increases. At capsule error probability $P_e = 0.01$, the system still can perform reasonably well, and is relatively insensitive to channel error. As the capsule error probability goes up to 0.1, a significant jump in both capsule loss and average capsule delay is observed. We believe that this is more due to congested channel than due to channel error.\(^2\)

![Graph showing capsule loss and delay with transmission error](image)

Figure 5.12: Modified DFQ performance in errored environment

The mixed traffic set at offered load of 12 Mbps or 0.48 of system capacity.

---

\(^2\)At capsule error probability of 0.9, the maximum system throughput for the mixed traffic set is $0.9 \times 0.524 = 0.472$. This is lower than offered load, which is 12 Mbps or 0.48 of channel capacity.
Performance vs number of remote stations  By intuition, the more remote stations there are, the worse overall system performance will be. Figure 5.13 interestingly shows that capsule loss most likely occurs when the number of remote stations is between 10 and 20. This observation can be explained by understanding the interaction between poll interval and delay requirements of services. When there are very few remote stations, the incoming traffic load of each remote station is so high that the likelihood of a remote queue becoming empty is slight. When the number of remote station increases, traffic load of each remote station becomes less dense, hence the chance that a remote queue become empty start to rise. With a remote station queue becomes empty, it must wait for 5 ms before the next data poll arrives. During this interval, there are many arrivals, especially video capsule arrivals since each station can have more than one video services. This in turn causes an instantaneous congestion at the remote station. As a result, some delay-sensitive capsules (mostly video capsules as they arrive in a much more bursty pattern than voice capsules) are time-expired and discarded. However as the number of remote stations goes up to over 20, capsule arrivals (especially video capsule arrivals) to each remote station becomes less dense. The arrivals during a 5 ms poll interval start to decrease, therefore instantaneous congestion at a remote station is less likely to occur and less number of capsules are discard.

Overall delay increases with number of remote stations. This is because an increase in remote station number leads to larger probability that remote queues become empty. Hence the likelihood that a capsule waits for additional time due to the poll interval also increases. This in turn leads to a higher average delay.
CHAPTER 5. SIMULATION RESULT ANALYSIS

Figure 5.13: Modified DFQ performance vs number of stations
The mixed traffic set at offered load of 12 Mbps or 0.48 of channel capacity.

5.3 Modified MDR-TDMA Simulation

5.3.1 Modified MDR-TDMA Simulation Scenarios and Parameters

MDR-TDMA does not allow direct remote-to-remote transmission. Any local-to-
local traffic must be transmitted via the base station. Thus traffic flows in MDR-
TDMA system are either downstream or upstream. As previously explained, the
MDR-TDMA system provides the largest maximum throughput, which is equal to maximum channel efficiency $\eta_{\text{max}}$, if the input traffic is external-to-local, and/or local-to-external. If all input traffic is local-to-local, the maximum throughput of system is cut to $0.5\eta_{\text{max}}$. In the MDR-TDMA simulation, input load consists of equal amount of external-to-local and local-to-external traffic. Table 5.6 lists the components of the input traffic. The performance in terms of system throughput, average delay, and capsule loss under various input loads is obtained from simulation. Also, simulations are run with different channel error probabilities in order to gain insight on the sensitivity of MDR-TDMA to transmission errors. Lastly, since we assign each remote station with a different request and control slot, random accesses are not required in MDR-TDMA. Hence the system performance will be relatively independent of the number of remote stations in the system.\(^3\)

<table>
<thead>
<tr>
<th>Input traffic</th>
<th>Medium Mixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>50% external-to-local</td>
<td>16.7% voice, 16.7% data, 16.7% video</td>
</tr>
<tr>
<td>50% local-to-external</td>
<td>16.7% voice, 16.7% data, 16.7% video</td>
</tr>
</tbody>
</table>

Table 5.6: MDR-TDMA simulation traffic mixes

MDR-TDMA simulation parameters are given in Table 5.7. All transmissions have taken guard time and synchronization time into account. Substituting these parameters into (3.5), maximum MDR-TDMA system throughput is $\eta_{\text{max}} = \text{thru}_{\text{max}} = 0.907 \times 0.690 = 0.63$, where 0.907 is the MAC efficiency $\eta_{\text{mac}}$, and 0.690 is the capsule efficiency $\eta_{\text{cap}}$. Note that the MAC efficiency has improved at least 10% over DFQ and modified DFQ. Even higher MAC efficiency can be achieved by increasing the superframe size. This is at the cost of increased average delay of capsules.

\(^3\)In fact, number of remote station changes the control slots overhead $T_{\text{control}}$ and request slot overhead $T_{\text{request}}$. But since the number of stations in the system is under 100, changing $T_{\text{control}}$ and $T_{\text{request}}$ has little effect on system performance.
### 5.3.2 Modified MDR-TDMA Simulation and Analysis

**Modified MDR-TDMA general performance** The overall system throughput of MDR-TDMA as a function of offered load is shown in Figure 5.14. The maximum throughput obtained from simulation is very close to the calculated result. Figure 5.15 and figure5.16 show capsule loss and average delay of voice, data, video traffic in upstream and downstream transmissions. The MDR-TDMA system starts to drop capsules when the offered load reaches approximately 0.56 of system capacity. That is, MDR-TDMA system offers QoS to services even when the total offered load is as high as 89% of maximum throughput. Even though the real system performance is subject to the traffic mixes of offered load, the simulation nonetheless provides an insight on how well modified MDR-TDMA can handle multimedia traffic.

The flat parts of delay curves in figure 5.16 reflect that under low to moderate load, capsules must wait in average more than half of the superframe length before being transmitted. Under heavy load, the average delay sharply increases with offered load because of channel congestion. Note also that the average delays of video and voice capsules in upstream (local-to-external) transmission is typically larger than the delay of their counterparts in downstream (external-to-local) traffic, while average delay of data capsules in upstream traffic is higher than that in downstream traffic. This is because upstream information slots are assigned to remote stations in partitions, therefore data capsules can be transmitted before video or voice capsules in an upstream subframe; whereas in downstream information slots, data capsules, for the most part, are transmitted after the voice or video capsules in the superframes

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Channel Bandwidth ($C$)</td>
<td>25Mbps</td>
</tr>
<tr>
<td>Tx/Rx turnaround time ($T_{\text{turnaround}}$)</td>
<td>2 µs</td>
</tr>
<tr>
<td>Superframe length ($T_{\text{frame}}$)</td>
<td>6 ms</td>
</tr>
<tr>
<td>Information slot length ($T_{c}$)</td>
<td>21 µs</td>
</tr>
<tr>
<td>Ack slots overhead ($T_{\text{ack}}$)</td>
<td>200 µs</td>
</tr>
<tr>
<td>Control slots overhead ($T_{\text{control}}$)</td>
<td>200 µs</td>
</tr>
<tr>
<td>Request slots overhead ($T_{\text{request}}$)</td>
<td>150 µs</td>
</tr>
</tbody>
</table>

Table 5.7: MDR-TDMA simulation parameters
as specified by their virtual time tags at base station.

Figure 5.14: Modified MDR-TDMA system throughput
Figure 5.15: Modified MDR-TDMA capsule loss
Voice traffic delay in modified MDR-TDMA

Data traffic delay in modified MDR-TDMA

Video traffic delay in modified MDR-TDMA

Figure 5.16: Modified MDR-TDMA average capsule delay

Data delay are presented in logarithm scale in order to cover average data delays at all input loads.
CHAPTER 5. SIMULATION RESULT ANALYSIS

Figure 5.17: Modified MDR-TDMA capsule distribution

Offered load is 14 Mbps or 0.56 of channel capacity. Traffic mix is specified in table 5.6.

Figure 5.17 illustrates the capsule delay distribution of the MDR-TDMA system. Due to its framed channel format, arriving capsules cannot be transmitted immediately regardless of its priority. Hence, most capsules experience 2-10 ms delay as shown in the figure.

Modified MDR-TDMA performance vs channel error probability Figure 5.18 illustrates MDR-TDMA performance in terms of overall average delay and capsule loss in an errored environment. As capsule error probability goes from $10^{-4}$
to $10^1$, a big jump is observed for both average delay and capsule loss. This is to be expected. MDR-TDMA is more vulnerable to bursty transmission errors than DFQ and modified DFQ are.

![Graphs showing capsule loss and delay with transmission error](image)

**Figure 5.18:** Modified MDR-TDMA performance in errored environment
Offered load is $14 \text{ Mbps}$ or $0.56$ of channel capacity. Traffic mix is specified in table 5.6.

Considering a bursty error period that covers the duration of the control slots, then remote stations will have no information regarding to the allocation of the next upstream subframe, and therefore not send anything during this upstream subframe. As a result, the upstream subframe is wasted even though the wireless channel might
have become clear again. Similarly, the base station relies on the acknowledgments from remote stations to determine whether downstream capsules have been successfully received. If the bursty error period covers all acknowledgment slots, the base station will assume that none of capsules sent in the previous downstream subframe is received. Therefore it will use the next downstream subframe to re-send these capsules, thus a downstream subframe is also wasted. This waste in system resources in turn translates into worse system performance. From the above analysis, it can be concluded that MDR-TDMA is more sensitive to burst channel errors than DFQ and modified DFQ, because channel error affects the information transmitted not only during the error period, but also the period following the error period.

5.4 Conclusions

So far we have analyzed the performances of DFQ, modified DFQ, and modified TDMA. The pros and cons of each protocol are listed in table 5.8. It appears that because of DFQ's need for exchanging virtual time information, the MAC overhead introduced has greatly undermined the system performance. This is especially the case if the number of remote stations in the system is more than a few.

Both modified DFQ and modified MDR-TDMA utilize the same scheduling scheme to provide good QoS, while keeping MAC overhead in bound. The choice between them will depend on the strengths and weaknesses of the underlying MAC protocol: polling based protocol or MDR-TDMA protocol. The polling based protocol is close to work-conserving, therefore average delay of capsules under light to moderate load is less than that in MDR-TDMA. In addition, the polling based protocol supports remote-to-remote transmission, this presents a major advantage over MDR-TDMA in terms system throughput if traffic in the system is largely local-to-local. The problem with the polling based protocol is large overhead associated with each capsule transmission, especially turn-around time. This overhead is hardware dependent. Given technology advances and/or adding modem complexity, it is feasible that turnaround time can be greatly reduced. This will result in a much smaller overhead in the polling based protocol, and make it more appealing. The main advantage of MDR-TDMA is that it requires a much smaller MAC overhead than the polling based protocol. As a result, MDR-TDMA channel efficiency is
higher than that for the polling based protocol. On the downside, MDR-TDMA is a non-work conserving system, and does not support remote-to-remote transmission. In addition, MDR-TDMA is more vulnerable to bursty channel errors.

Table 5.8: Summary of protocol strengths and weaknesses

<table>
<thead>
<tr>
<th>Protocol</th>
<th>DFQ</th>
<th>Modified DFQ</th>
<th>Modified MDR-TDMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>QoS support</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Maximum throughput</td>
<td>bad</td>
<td>good, if traffic</td>
<td>good, if little traffic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>largely local-to-local</td>
<td>local-to-local traffic</td>
</tr>
<tr>
<td>Support for many remote stations</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Remote-to-remote transmission</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Sensitivity to bursty channel error</td>
<td>not sensitive</td>
<td>not sensitive</td>
<td>sensitive</td>
</tr>
</tbody>
</table>
Appendix A

Confidence Interval

90% confidence intervals are calculated for all simulation results, and are listed in the following tables. \( \mu \) is estimated mean delay or capsule loss ratio obtained from averaging the results from five independent simulation runs, \( C_{int} \) is the confidence interval associated with each \( \mu \).

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu ) (ms)</td>
<td>( C_{int} ) (ms)</td>
<td>( \mu ) (ms)</td>
<td>( C_{int} ) (ms)</td>
</tr>
<tr>
<td>0.16</td>
<td>0.0298 ±0.000071</td>
<td>0.031 ±0.000044</td>
<td>0.0292 ±0.000020</td>
</tr>
<tr>
<td>0.24</td>
<td>0.0346 ±0.000166</td>
<td>0.0381 ±0.000071</td>
<td>0.0310 ±0.000021</td>
</tr>
<tr>
<td>0.32</td>
<td>0.0462 ±0.000290</td>
<td>0.0740 ±0.00202</td>
<td>0.0353 ±0.000044</td>
</tr>
<tr>
<td>0.40</td>
<td>0.0823 ±0.000751</td>
<td>0.456 ±0.0179</td>
<td>0.0425 ±0.000090</td>
</tr>
<tr>
<td>0.48</td>
<td>0.0280 ±0.00542</td>
<td>5.372 ±0.296</td>
<td>0.0597 ±0.000307</td>
</tr>
<tr>
<td>0.56</td>
<td>6.283 ±0.0460</td>
<td>223.22 ±2.395</td>
<td>1.634 ±0.0333</td>
</tr>
<tr>
<td>0.60</td>
<td>9.105 ±0.0156</td>
<td>317.88 ±1.346</td>
<td>5.618 ±0.0433</td>
</tr>
<tr>
<td>0.64</td>
<td>10.91 ±0.0293</td>
<td>372.73 ±0.749</td>
<td>6.985 ±0.0185</td>
</tr>
</tbody>
</table>

Table A.1: DFQ average delay confidence interval: all external traffic set
### Table A.2: DFQ capsule loss confidence interval: all external traffic set

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$</td>
<td>$C_{int}$</td>
<td>$\mu$</td>
</tr>
<tr>
<td>0.16</td>
<td>0</td>
<td>$\pm0$</td>
<td>0</td>
</tr>
<tr>
<td>0.24</td>
<td>0</td>
<td>$\pm0$</td>
<td>0</td>
</tr>
<tr>
<td>0.32</td>
<td>0</td>
<td>$\pm0$</td>
<td>0</td>
</tr>
<tr>
<td>0.40</td>
<td>0</td>
<td>$\pm0$</td>
<td>0</td>
</tr>
<tr>
<td>0.48</td>
<td>0</td>
<td>$\pm0$</td>
<td>0</td>
</tr>
<tr>
<td>0.56</td>
<td>0</td>
<td>$\pm0$</td>
<td>0.154</td>
</tr>
<tr>
<td>0.60</td>
<td>0</td>
<td>$\pm0$</td>
<td>0.332</td>
</tr>
<tr>
<td>0.64</td>
<td>0</td>
<td>$\pm0$</td>
<td>0.467</td>
</tr>
</tbody>
</table>

### Table A.3: DFQ average delay confidence interval: all local traffic set

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$ (ms)</td>
<td>$C_{int}$ (ms)</td>
<td>$\mu$ (ms)</td>
</tr>
<tr>
<td>0.16</td>
<td>3.473</td>
<td>$\pm0.00470$</td>
<td>125.535</td>
</tr>
<tr>
<td>0.24</td>
<td>5.658</td>
<td>$\pm0.00392$</td>
<td>211.46</td>
</tr>
<tr>
<td>0.32</td>
<td>9.014</td>
<td>$\pm0.0174$</td>
<td>358.39</td>
</tr>
<tr>
<td>0.40</td>
<td>14.537</td>
<td>$\pm0.0859$</td>
<td>457.918</td>
</tr>
<tr>
<td>0.48</td>
<td>19.679</td>
<td>$\pm0.0182$</td>
<td>479.789</td>
</tr>
<tr>
<td>0.56</td>
<td>20.837</td>
<td>$\pm0.00523$</td>
<td>514.654</td>
</tr>
<tr>
<td>0.60</td>
<td>20.181</td>
<td>$\pm0.00490$</td>
<td>536.902</td>
</tr>
</tbody>
</table>

### Table A.4: DFQ capsule loss confidence interval: all local traffic set

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$</td>
<td>$C_{int}$</td>
<td>$\mu$</td>
</tr>
<tr>
<td>0.16</td>
<td>0</td>
<td>$\pm0$</td>
<td>0.0542</td>
</tr>
<tr>
<td>0.24</td>
<td>0</td>
<td>$\pm0$</td>
<td>0.121</td>
</tr>
<tr>
<td>0.32</td>
<td>0</td>
<td>$\pm0$</td>
<td>0.411</td>
</tr>
<tr>
<td>0.40</td>
<td>0.0121</td>
<td>$\pm0.00251$</td>
<td>0.592</td>
</tr>
<tr>
<td>0.48</td>
<td>0.495</td>
<td>$\pm0.00126$</td>
<td>0.757</td>
</tr>
<tr>
<td>0.56</td>
<td>0.779</td>
<td>$\pm0.00105$</td>
<td>0.777</td>
</tr>
<tr>
<td>0.60</td>
<td>0.842</td>
<td>$\pm0.000937$</td>
<td>0.764</td>
</tr>
</tbody>
</table>
### Table A.5: DFQ average delay confidence interval: mix traffic set

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$ (ms)</td>
<td>$C_{int}$ (ms)</td>
<td>$\mu$ (ms)</td>
</tr>
<tr>
<td>0.16</td>
<td>2.597 ±0.00276</td>
<td>95.577 ±1.918</td>
<td>0.163 ±0.000235</td>
</tr>
<tr>
<td>0.24</td>
<td>3.820 ±0.00327</td>
<td>143.086 ±1.423</td>
<td>0.222 ±0.000091</td>
</tr>
<tr>
<td>0.32</td>
<td>6.665 ±0.0103</td>
<td>256.84 ±0.496</td>
<td>1.378 ±0.0137</td>
</tr>
<tr>
<td>0.40</td>
<td>9.039 ±0.00147</td>
<td>352.423 ±1.661</td>
<td>5.385 ±0.0306</td>
</tr>
<tr>
<td>0.48</td>
<td>16.720 ±0.0928</td>
<td>462.771 ±1.496</td>
<td>9.161 ±0.00553</td>
</tr>
<tr>
<td>0.56</td>
<td>19.284 ±0.0181</td>
<td>478.239 ±1.361</td>
<td>9.517 ±0.00288</td>
</tr>
<tr>
<td>0.60</td>
<td>18.489 ±0.0164</td>
<td>480.667 ±0.652</td>
<td>9.215 ±0.00104</td>
</tr>
<tr>
<td>0.64</td>
<td>19.983 ±0.0101</td>
<td>495.804 ±0.665</td>
<td>8.894 ±0.00223</td>
</tr>
</tbody>
</table>

### Table A.6: DFQ average loss confidence interval: mix traffic set

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$</td>
<td>$C_{int}$</td>
<td>$\mu$</td>
</tr>
<tr>
<td>0.16</td>
<td>0 ±0</td>
<td>0.0373 ±0.00260</td>
<td>0 ±0</td>
</tr>
<tr>
<td>0.24</td>
<td>0 ±0</td>
<td>0.0666 ±0.00439</td>
<td>0 ±0</td>
</tr>
<tr>
<td>0.32</td>
<td>0 ±0</td>
<td>0.169 ±0.00408</td>
<td>0 ±0</td>
</tr>
<tr>
<td>0.40</td>
<td>0 ±0</td>
<td>0.398 ±0.00252</td>
<td>0.000608 ±0.000101</td>
</tr>
<tr>
<td>0.48</td>
<td>0.0264 ±0.00076</td>
<td>0.713 ±0.0011</td>
<td>0.00654 ±0.000608</td>
</tr>
<tr>
<td>0.56</td>
<td>0.314 ±0.00238</td>
<td>0.741 ±0.00405</td>
<td>0.087 ±0.000249</td>
</tr>
<tr>
<td>0.60</td>
<td>0.531 ±0.00112</td>
<td>0.754 ±0.00173</td>
<td>0.115 ±0.000067</td>
</tr>
<tr>
<td>0.64</td>
<td>0.662 ±0.00165</td>
<td>0.766 ±0.001</td>
<td>0.160 ±0.000409</td>
</tr>
</tbody>
</table>

### Table A.7: DFQ confidence interval: transmission error

<table>
<thead>
<tr>
<th>Trans. error</th>
<th>Delay</th>
<th>Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$ (ms)</td>
<td>$C_{int}$ (ms)</td>
</tr>
<tr>
<td>0.0001</td>
<td>58.10 ±0.21</td>
<td>0.267 ±0.000533</td>
</tr>
<tr>
<td>0.001</td>
<td>58.10 ±0.21</td>
<td>0.268 ±0.000534</td>
</tr>
<tr>
<td>0.01</td>
<td>58.17 ±0.22</td>
<td>0.276 ±0.000673</td>
</tr>
<tr>
<td>0.02</td>
<td>58.26 ±0.214</td>
<td>0.286 ±0.00103</td>
</tr>
<tr>
<td>0.05</td>
<td>58.82 ±0.144</td>
<td>0.319 ±0.000797</td>
</tr>
<tr>
<td>0.1</td>
<td>59.86 ±0.301</td>
<td>0.376 ±0.00194</td>
</tr>
</tbody>
</table>
## Table A.8: DFQ confidence interval: number of stations

<table>
<thead>
<tr>
<th>Num of stations</th>
<th>Delay $\mu$ (ms)</th>
<th>$C_{int}$ (ms)</th>
<th>Loss $\mu$</th>
<th>$C_{int}$ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>85.47</td>
<td>±0.667</td>
<td>0.0863</td>
<td>±0.00118</td>
</tr>
<tr>
<td>10</td>
<td>82.32</td>
<td>±0.245</td>
<td>0.167</td>
<td>±0.00095</td>
</tr>
<tr>
<td>20</td>
<td>58.80</td>
<td>±0.262</td>
<td>0.262</td>
<td>±0.00055</td>
</tr>
<tr>
<td>30</td>
<td>58.62</td>
<td>±0.263</td>
<td>0.263</td>
<td>±0.00053</td>
</tr>
<tr>
<td>40</td>
<td>58.36</td>
<td>±0.229</td>
<td>0.265</td>
<td>±0.00056</td>
</tr>
<tr>
<td>50</td>
<td>58.10</td>
<td>±0.212</td>
<td>0.267</td>
<td>±0.00050</td>
</tr>
<tr>
<td>60</td>
<td>57.93</td>
<td>±0.220</td>
<td>0.270</td>
<td>±0.00042</td>
</tr>
<tr>
<td>70</td>
<td>57.79</td>
<td>±0.222</td>
<td>0.273</td>
<td>±0.00039</td>
</tr>
</tbody>
</table>

## Table A.9: Modified DFQ average delay confidence interval: all external traffic set

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice $\mu$ (ms)</th>
<th>$C_{int}$ (ms)</th>
<th>Data $\mu$ (ms)</th>
<th>$C_{int}$ (ms)</th>
<th>Video $\mu$ (ms)</th>
<th>$C_{int}$ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.16</td>
<td>0.0283</td>
<td>±0.000063</td>
<td>0.0278</td>
<td>±0.000039</td>
<td>0.0278</td>
<td>±0.000020</td>
</tr>
<tr>
<td>0.24</td>
<td>0.0325</td>
<td>±0.00015</td>
<td>0.0294</td>
<td>±0.000066</td>
<td>0.0294</td>
<td>±0.000022</td>
</tr>
<tr>
<td>0.32</td>
<td>0.0419</td>
<td>±0.00022</td>
<td>0.0588</td>
<td>±0.00141</td>
<td>0.0332</td>
<td>±0.000047</td>
</tr>
<tr>
<td>0.40</td>
<td>0.0682</td>
<td>±0.000466</td>
<td>0.264</td>
<td>±0.00999</td>
<td>0.0395</td>
<td>±0.000067</td>
</tr>
<tr>
<td>0.48</td>
<td>0.194</td>
<td>±0.00284</td>
<td>2.846</td>
<td>±0.0717</td>
<td>0.0538</td>
<td>±0.000275</td>
</tr>
<tr>
<td>0.56</td>
<td>2.875</td>
<td>±0.123</td>
<td>100.343</td>
<td>±4.592</td>
<td>0.286</td>
<td>±0.0199</td>
</tr>
<tr>
<td>0.60</td>
<td>7.814</td>
<td>±0.0205</td>
<td>275.682</td>
<td>±1.830</td>
<td>3.865</td>
<td>±0.0582</td>
</tr>
<tr>
<td>0.64</td>
<td>9.80</td>
<td>±0.0142</td>
<td>346.086</td>
<td>±0.585</td>
<td>6.202</td>
<td>±0.0331</td>
</tr>
</tbody>
</table>
### Table A.10: Modified DFQ capsule loss confidence interval: all external traffic set

<table>
<thead>
<tr>
<th>Offered load</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \mu )</td>
<td>( C_{\text{int}} )</td>
<td>( \mu )</td>
</tr>
<tr>
<td>( 0.16 )</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>( 0.24 )</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>( 0.32 )</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>( 0.40 )</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>( 0.48 )</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>( 0.56 )</td>
<td>0 ± 0</td>
<td>0.0459 ± 0.00515</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>( 0.60 )</td>
<td>0 ± 0</td>
<td>0.241 ± 0.00437</td>
<td>0.0144 ± 0.000312</td>
</tr>
<tr>
<td>( 0.64 )</td>
<td>0 ± 0</td>
<td>0.391 ± 0.00433</td>
<td>0.0100 ± 0.000222</td>
</tr>
</tbody>
</table>

### Table A.11: Modified DFQ average delay confidence interval: all local traffic set

<table>
<thead>
<tr>
<th>Offered load</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \mu ) (ms)</td>
<td>( C_{\text{int}} ) (ms)</td>
<td>( \mu ) (ms)</td>
</tr>
<tr>
<td>( 0.16 )</td>
<td>2.551 ± 0.00325</td>
<td>2.603 ± 0.00175</td>
<td>2.541 ± 0.00210</td>
</tr>
<tr>
<td>( 0.24 )</td>
<td>2.565 ± 0.0007</td>
<td>2.644 ± 0.00240</td>
<td>2.528 ± 0.00109</td>
</tr>
<tr>
<td>( 0.32 )</td>
<td>2.586 ± 0.00204</td>
<td>2.911 ± 0.0115</td>
<td>2.509 ± 0.00189</td>
</tr>
<tr>
<td>( 0.40 )</td>
<td>2.648 ± 0.00055</td>
<td>5.219 ± 0.0438</td>
<td>2.585 ± 0.0176</td>
</tr>
<tr>
<td>( 0.48 )</td>
<td>3.475 ± 0.0362</td>
<td>39.721 ± 1.972</td>
<td>7.797 ± 0.0848</td>
</tr>
<tr>
<td>( 0.56 )</td>
<td>8.195 ± 0.0520</td>
<td>115.34 ± 2.493</td>
<td>10.308 ± 0.0517</td>
</tr>
<tr>
<td>( 0.60 )</td>
<td>8.689 ± 0.0250</td>
<td>172.36 ± 1.082</td>
<td>9.194 ± 0.0402</td>
</tr>
<tr>
<td>( 0.64 )</td>
<td>9.797 ± 0.0742</td>
<td>210.66 ± 3.968</td>
<td>7.909 ± 0.0424</td>
</tr>
</tbody>
</table>

### Table A.12: Modified DFQ capsule loss confidence interval: all local traffic set

<table>
<thead>
<tr>
<th>Offered load</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \mu )</td>
<td>( C_{\text{int}} )</td>
<td>( \mu )</td>
</tr>
<tr>
<td>( 0.16 )</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>( 0.24 )</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>( 0.32 )</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>( 0.40 )</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>( 0.48 )</td>
<td>0.000098 ± 0.000067</td>
<td>0.00320 ± 0.00177</td>
<td>0.0374 ± 0.00273</td>
</tr>
<tr>
<td>( 0.56 )</td>
<td>0.0521 ± 0.00149</td>
<td>0.0940 ± 0.00262</td>
<td>0.275 ± 0.00407</td>
</tr>
<tr>
<td>( 0.60 )</td>
<td>0.0986 ± 0.00184</td>
<td>0.157 ± 0.00383</td>
<td>0.341 ± 0.00228</td>
</tr>
<tr>
<td>( 0.64 )</td>
<td>0.135 ± 0.0023</td>
<td>0.211 ± 0.00269</td>
<td>0.396 ± 0.00185</td>
</tr>
</tbody>
</table>
## APPENDIX A. CONFIDENCE INTERVAL

### Table A.13: Modified DFQ average delay confidence interval: mix traffic set

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$ (ms)</td>
<td>$C_{int}$ (ms)</td>
<td>$\mu$ (ms)</td>
</tr>
<tr>
<td>0.16</td>
<td>1.706 ±0.00180</td>
<td>1.737 ±0.00144</td>
<td>1.696 ±0.00145</td>
</tr>
<tr>
<td>0.24</td>
<td>1.718 ±0.00210</td>
<td>1.754 ±0.00123</td>
<td>1.706 ±0.00134</td>
</tr>
<tr>
<td>0.32</td>
<td>1.741 ±0.00050</td>
<td>1.897 ±0.00357</td>
<td>1.698 ±0.000489</td>
</tr>
<tr>
<td>0.40</td>
<td>1.769 ±0.00114</td>
<td>2.446 ±0.0190</td>
<td>1.717 ±0.00212</td>
</tr>
<tr>
<td>0.48</td>
<td>2.165 ±0.00902</td>
<td>13.571 ±0.945</td>
<td>3.192 ±0.0409</td>
</tr>
<tr>
<td>0.56</td>
<td>5.481 ±0.0828</td>
<td>99.983 ±1.832</td>
<td>7.694 ±0.0203</td>
</tr>
<tr>
<td>0.60</td>
<td>7.674 ±0.0759</td>
<td>154.998 ±1.560</td>
<td>7.310 ±0.0197</td>
</tr>
<tr>
<td>0.64</td>
<td>9.720 ±0.0518</td>
<td>224.540 ±2.823</td>
<td>8.339 ±0.0336</td>
</tr>
</tbody>
</table>

### Table A.14: Modified DFQ capsule loss confidence interval: mix traffic set

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$</td>
<td>$C_{int}$</td>
<td>$\mu$</td>
</tr>
<tr>
<td>0.16</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.24</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.32</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.40</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.48</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.56</td>
<td>0.0213</td>
<td>±0.00115</td>
<td>0.0332</td>
</tr>
<tr>
<td>0.60</td>
<td>0.0946</td>
<td>±0.00289</td>
<td>0.0963</td>
</tr>
<tr>
<td>0.64</td>
<td>0.132</td>
<td>±0.00163</td>
<td>0.195</td>
</tr>
</tbody>
</table>

### Table A.15: Modified DFQ confidence interval: transmission error

<table>
<thead>
<tr>
<th>Trans. error</th>
<th>Delay</th>
<th>Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$ (ms)</td>
<td>$C_{int}$ (ms)</td>
</tr>
<tr>
<td>0.0001</td>
<td>6.307</td>
<td>±0.390</td>
</tr>
<tr>
<td>0.001</td>
<td>6.388</td>
<td>±0.340</td>
</tr>
<tr>
<td>0.01</td>
<td>7.406</td>
<td>±0.550</td>
</tr>
<tr>
<td>0.02</td>
<td>8.743</td>
<td>±0.628</td>
</tr>
<tr>
<td>0.05</td>
<td>15.074</td>
<td>±0.887</td>
</tr>
<tr>
<td>0.1</td>
<td>33.382</td>
<td>±1.437</td>
</tr>
</tbody>
</table>
### Table A.16: Modified DFQ confidence interval: number of stations

<table>
<thead>
<tr>
<th>Num of station</th>
<th>Delay</th>
<th>Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \mu (\text{ms}) )</td>
<td>( C_{\text{int}} (\text{ms}) )</td>
</tr>
<tr>
<td>5</td>
<td>5.438</td>
<td>±0.248</td>
</tr>
<tr>
<td>10</td>
<td>4.796</td>
<td>±0.163</td>
</tr>
<tr>
<td>20</td>
<td>4.746</td>
<td>±0.146</td>
</tr>
<tr>
<td>30</td>
<td>5.165</td>
<td>±0.221</td>
</tr>
<tr>
<td>40</td>
<td>5.634</td>
<td>±0.247</td>
</tr>
<tr>
<td>50</td>
<td>6.278</td>
<td>±0.349</td>
</tr>
<tr>
<td>60</td>
<td>7.215</td>
<td>±0.506</td>
</tr>
<tr>
<td>70</td>
<td>8.165</td>
<td>±0.518</td>
</tr>
</tbody>
</table>

### Table A.17: Modified TDMA average delay confidence interval: uplink

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \mu (\text{ms}) )</td>
<td>( C_{\text{int}} (\text{ms}) )</td>
<td>( \mu (\text{ms}) )</td>
</tr>
<tr>
<td>0.16</td>
<td>3.182</td>
<td>±0.00314</td>
<td>3.855</td>
</tr>
<tr>
<td>0.24</td>
<td>3.979</td>
<td>±0.00528</td>
<td>4.034</td>
</tr>
<tr>
<td>0.32</td>
<td>4.197</td>
<td>±0.00408</td>
<td>4.266</td>
</tr>
<tr>
<td>0.40</td>
<td>4.360</td>
<td>±0.00398</td>
<td>4.793</td>
</tr>
<tr>
<td>0.48</td>
<td>4.614</td>
<td>±0.00451</td>
<td>7.729</td>
</tr>
<tr>
<td>0.56</td>
<td>4.762</td>
<td>±0.00582</td>
<td>16.074</td>
</tr>
<tr>
<td>0.60</td>
<td>5.610</td>
<td>±0.0431</td>
<td>59.463</td>
</tr>
<tr>
<td>0.64</td>
<td>7.691</td>
<td>±0.0705</td>
<td>120.524</td>
</tr>
</tbody>
</table>
### Table A.18: Modified TDMA capsule loss confidence interval: uplink

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$</td>
<td>$C_{int}$</td>
<td>$\mu$</td>
</tr>
<tr>
<td>0.16</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.24</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.32</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.40</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.48</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.56</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.60</td>
<td>0.000952</td>
<td>±0.000147</td>
<td>0.00526</td>
</tr>
<tr>
<td>0.64</td>
<td>0.0155</td>
<td>±0.000425</td>
<td>0.0382</td>
</tr>
</tbody>
</table>

### Table A.19: Modified TDMA average delay confidence interval: downlink

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$ (ms)</td>
<td>$C_{int}$ (ms)</td>
<td>$\mu$ (ms)</td>
</tr>
<tr>
<td>0.16</td>
<td>3.390</td>
<td>±0.00461</td>
<td>3.749</td>
</tr>
<tr>
<td>0.24</td>
<td>3.555</td>
<td>±0.00253</td>
<td>4.408</td>
</tr>
<tr>
<td>0.32</td>
<td>3.745</td>
<td>±0.00145</td>
<td>4.384</td>
</tr>
<tr>
<td>0.40</td>
<td>3.891</td>
<td>±0.00146</td>
<td>5.017</td>
</tr>
<tr>
<td>0.48</td>
<td>4.140</td>
<td>±0.00263</td>
<td>7.528</td>
</tr>
<tr>
<td>0.56</td>
<td>4.568</td>
<td>±0.0192</td>
<td>32.654</td>
</tr>
<tr>
<td>0.60</td>
<td>5.161</td>
<td>±0.0382</td>
<td>88.872</td>
</tr>
<tr>
<td>0.64</td>
<td>8.163</td>
<td>±0.104</td>
<td>224.17</td>
</tr>
</tbody>
</table>

### Table A.20: Modified TDMA capsule loss confidence interval: downlink

<table>
<thead>
<tr>
<th>Offered load frac. of 25 Mbps</th>
<th>Voice</th>
<th>Data</th>
<th>Video</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$</td>
<td>$C_{int}$</td>
<td>$\mu$</td>
</tr>
<tr>
<td>0.16</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.24</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.32</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.40</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.48</td>
<td>0</td>
<td>±0</td>
<td>0</td>
</tr>
<tr>
<td>0.56</td>
<td>0</td>
<td>±0</td>
<td>0.000621</td>
</tr>
<tr>
<td>0.60</td>
<td>0</td>
<td>±0</td>
<td>0.0349</td>
</tr>
<tr>
<td>0.64</td>
<td>0</td>
<td>±0</td>
<td>0.143</td>
</tr>
</tbody>
</table>
Table A.21: Modified TDMA confidence interval: transmission error

<table>
<thead>
<tr>
<th>Trans. error</th>
<th>Delay</th>
<th>Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$ (ms)</td>
<td>$C_{int}$ (ms)</td>
</tr>
<tr>
<td>0.0001</td>
<td>28.875</td>
<td>±0.724</td>
</tr>
<tr>
<td>0.001</td>
<td>29.673</td>
<td>±0.800</td>
</tr>
<tr>
<td>0.01</td>
<td>38.391</td>
<td>±0.858</td>
</tr>
<tr>
<td>0.02</td>
<td>47.289</td>
<td>±1.126</td>
</tr>
<tr>
<td>0.05</td>
<td>71.269</td>
<td>±1.119</td>
</tr>
<tr>
<td>0.1</td>
<td>90.737</td>
<td>±0.668</td>
</tr>
</tbody>
</table>
Bibliography

[1] ETS 300 652. Radio Equipment and Systems (RES); Hlgh PErformance Ra-
dio Local Area Network (HIPERLAN) Type 1; Functional specification. ETSI, October 1992.


[4] Federal Communications Commission. FCC 97-005, Amendment of the com-
mision’s rules to provide for operations of unlicensed NII devices in the 5 GHz frequency range. FCC, 1997.


[8] P. Goyal, S. Lam, and H. Vin. Determine end-to-end delay bounds in hetero-


M"aisie programming language. UCLA.


