C-Flow: A Compiler for Statically Scheduled Message Passing in Parallel Programs

by

Patrick Griffin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005 [June 2005]

Author .....
Department of Electrical Engineering and Computer Science
May 9, 2005

Certified by .....
Anant Agarwal
Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

[redacted]

C-Flow: A Compiler for Statically Scheduled Message Passing in Parallel Programs

by

Patrick Griffin

Submitted to the Department of Electrical Engineering and Computer Science
on May 9, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Performance improvement in future microprocessors will rely more on the exploitation of parallelism than increases in clock frequency, leading to more multi-core and tiled processor architectures. Despite continuing research into parallelizing compilers, programming multiple instruction stream architectures remains difficult. This document describes C-Flow, a compiler system enabling statically-scheduled message passing between programs running on separate processors.

When combined with statically-scheduled, low-latency networks like those in the MIT Raw processor, C-Flow provides the programmer with a simple but comprehensive messaging interface that can be used from high-level languages like C. The use of statically-scheduled messaging allows for fine-grained (single-word) messages that would be quite inefficient in the more traditional message passing systems used in cluster computers. Such fine-grained parallelism is possible because, as in systolic array machines, the network provides all of the necessary synchronization between tiles. On the Raw processor, C-Flow reduces development complexity by allowing the programmer to schedule static messages from a high-level language instead of using assembly code. C-Flow programs have been developed for arrays with 64 or more processor tiles and hve demonstrated performance within twenty percent of hand-optimized assembly.

Thesis Supervisor: Anant Agarwal
Title: Professor

Chapter 1

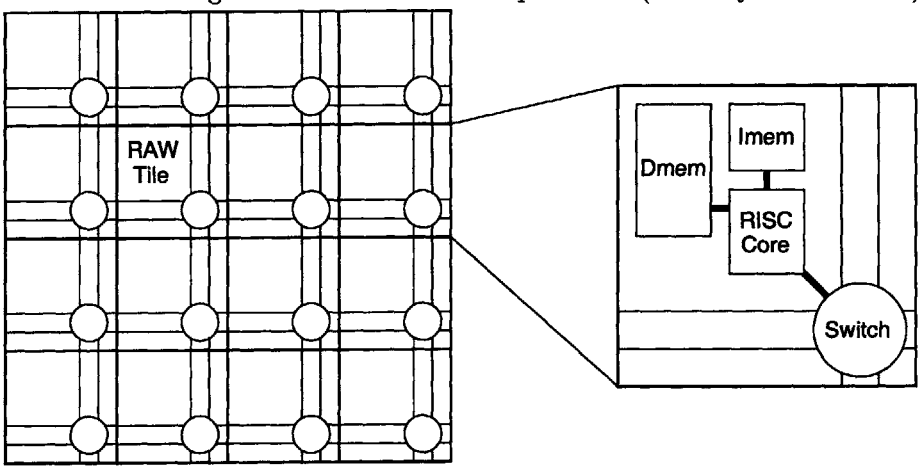
Introduction

Over the last several decades, the rapid development of microelectronics technology has allowed computer architects to design machines with ever-increasing numbers of smaller and faster transistors. In the last few years, though, the ways in which architects can use these resources have changed. While transistors continue to shrink, becoming faster and less power-hungry, the wires that connect them have not scaled as nicely; we are now entering an era in which the cost of sending signals across wires is the predominant factor in determining a microprocessor's speed and power usage. As a result, new architectures have chosen to improve performance through parallelism instead of frequency scaling. Instead of spending silicon resources on ever-larger superscalar machines, architects have begun tiling multiple processors onto a single piece of silicon.

The MIT Raw processor is one such "tiled architecture". Its 16 cores can run 16 separate programs, taking advantage of the thread-level parallelism that Intel and others are pursuing in upcoming chip multiprocessor designs. However, Raw can also connect all 16 cores via a low-latency scalar operand network [1], using all of the distributed resources to work on a single program. Using this approach, Raw can also take advantage of instruction-level parallelism, as well provide excellent performance on stream programs[2].

However, writing programs that take maximum advantage of the scalar operand network requires careful tuning of network communication. Parallelizing compilers do an effective job of extracting instruction level parallelism, but writing efficient stream programs has required hand-coded assembly to route data over the operand network. This document describes C-Flow, a system for automatically generating the network routing code needed

Figure 1-1: The Raw Microprocessor (courtesy Jason Miller)



to communicate between Raw tiles. C-Flow allows the programmer to write a simple C program for each tile, specifying any data that should be sent to or received from other tiles. The C-Flow router then generates the necessary routing code, attaining performance on the order of that generated by hand-coded assembly while avoiding the arduous programming process.

1.1 The Raw Processor

The C-Flow compiler chain targets the MIT Raw processor. The Raw processor is composed of 16 symmetrical tiles, arranged into a 4-by-4 grid. Larger grids can be formed by placing multiple Raw chips next to each other, potentially creating grids with 64 or more tiles. Each tile is actually composed of two processors, one for data processing and the other for communication. The following sections describe the data and communication processors that make up each tile; a graphical representation appears in Figure 1.1 and a complete specification of the Raw processor is given in [3].

1.1.1 The Compute Processor

Each Raw tile contains a single-issue, pipelined compute processor. This processor contains a 32-entry register file, integer and floating arithmetic units, and a load/store unit for communicating with main memory. Instruction and data memory caches are also included, each with 32 KB of storage. The compute processor's ISA closely resembles that of a MIPS

R5000. Thus, from a very high level, Raw could be viewed as a collection of 16 in-order RISC processors placed on the same die.

However, the compute processor does have a few unique characteristics that are useful when parallelizing a program across multiple tiles. Of particular interest is the register-mapped interface to the scalar operand network. Two register numbers, 24 and 26, have been reserved for communicating data to and from the network. Any operation that reads from these registers will pull its operand off of the network via the switch communication processor; similarly, any operation that writes to them will send its output to the switch processor. Thus, reading from and writing to registers 24 or 26 allows the current tile program to communicate with any other tile program via the switch processor.

1.1.2 The Switch Processor

Raw's scalar operand network is implemented as two identical statically-scheduled networks. Arranged in a grid topology, these two "static networks" allow a tile to communicate with its neighbors to the north, south, east, and west. Data words are routed in these four directions, as well as to or from the compute processor, according to an instruction stream running on the switch processor. The switch processor ISA is a 64-bit VLIW form, encoded into two portions: the control flow instruction and the routing instruction.

The switch processor's routing component specifies a data source for each of the possible outgoing ports. For each instruction, the switch processor waits until data is available on all incoming ports that the instruction specifies should be routed to some outgoing port. When all of the needed data is available, the switch processor uses a crossbar to route all of the specified data words in a single cycle, then advances to the next instruction.

The address of that next instruction is determined by the control flow component of the switch VLIW instruction. The control flow component controls a scalar processor which has four registers and is capable of executing branches, jumps, and no-ops. On a no-op, the switch processor will advance to the next instruction once data routing has been completed; on jump or branch instructions, the next instruction is specified via the control flow instruction component. The scalar control flow processor allows the switch processor to match the control flow on the compute processor. If the control flow is properly matched, the compute processor will sync up with the switch processor, allowing the main tile program to send and receive data as needed via the switch processor controlled static networks.

1.1.3 Comparing Static and Dynamic Networks

Raw's static networks allow separate tiles to communicate with exceptionally low latency, but do so by introducing a new class of networks. Traditional multiprocessor systems have used "dynamically routed networks", in which messages are routed between nodes using information embedded in a "message header". A message header is shipped along with the data words being sent in each message; each time a message arrives at a node, the router examines the header and determines whether the message has arrived at its destination or where the message should be forwarded. Using a dynamic routing scheme allows greater temporal flexibility in routing - because all of the information needed to determine a message's destination is included within the header, the message can be sent at any time. In particular, the routers along a message path need not have any special information in advance of the message's arrival.

In Raw's statically scheduled networks, router circuits are replaced with switch processors. Messages are transmitted across the chip by a sequence of switches; each switch is responsible for moving the message data across a single communication link. Consequently, all switch processors on a message's path must execute an individual instruction to move the message from one link to the next. Thus, the compiler or programmer must know the scheduling of all potential messages in advance in order to schedule the individual routing instructions on the switch processors.

While dynamically routed networks offer greater flexibility, easier scheduling, and simple handling of dynamic events like interrupts, statically scheduled networks offer several advantages for certain classes of applications. First, Raw's statically scheduled networks do not require header words, so no network bandwidth is wasted transmitting extra routing information. Secondly, because all the routing information is specified in the switch processor's instruction streams, the switch processors can prepare routing operations before the message data actually arrives, allowing for lower message transmission latency. Finally, Raw's static network implementation allows a single data word to be sent to multiple ports, creating a simple mechanism for sending broadcast-type messages. Whereas many dynamic networks would implement broadcasts as multiple messages, one for each recipient, Raw's static network allows a single tile to broadcast to an arbitrary set of recipients with only a single send operation. Thus, static networks sacrifice the ability to handle dynamic events

in exchange for better bandwidth utilization, lower latency, and an easier mechanism for sending messages to multiple recipients. C-Flow allows the programmer to take advantage of these network characteristics without having to program the individual switch processors.

1.2 Related Work

At present, there a several methods by which a programmer can generate the compute and switch processor code needed to execute a program on Raw. These methods include RawCC, a parallelizing compiler that generates multiple tile programs from a single sequential C program, StreamIt, a language for specifying linear filter-based programs, and hand coding. These mechanisms have been used to show that Raw provides performance that ranges from comparable to orders of magnitude better than that of a Pentium 3 implemented on the same process[2].

Parallelizing compilers like RawCC [4] and the IMPACT-based [5] RepTile present the most familiar interface for programming Raw. These compiler tools take a single sequential C-language program as input and transform it into separate programs for each tile. The tile programs communicate via the static network; by working together, Raw's 16 tiles can contribute their execution resources to completing a single program. While our parallelizing compiler tools have successfully compiled many SpecInt benchmarks, these tools remain rather fragile. Using them requires careful attention to the compiler environment, and proper compilation is not guaranteed. Additionally, these tools provide adequate performance only when used to extract instruction-level parallelism. If a program can been structured as a stream of data flowing through filters, other programming methods can achieve much higher performance.

The high-level StreamIt language has been developed for programs that do fit this "streaming" model [6]. StreamIt programs are specified as a set of filters operating on streams of data. This compiler system provides an exceptionally clear method for programming linear filters. Even better, the performance of its generated programs has been equal to that of hand-coded programs in several cases. However, StreamIt is only useful for programs that can be described as linear filters. Many programs do not fit into this paradigm; for instance, algorithms that make use of pointer-based data structures (graphs, trees, etc.) are not a good match for StreamIt.

Hand-coded programs offer the best potential for performance optimization. Hand-coding requires writing tile programs in C or assembly. The routing code for the switch processor must be written in assembly. The later task can be particularly arduous, as it requires careful attention to synchronization and deadlock avoidance. However, in cases where the programmer has a clear idea of how the program should be partitioned across Raw's tiles, hand coding is the clear performance winner. Hand-coded streaming algorithms have shown remarkable scaling properties, in some cases actually improving their percentage resource utilization as the number of tiles is increased [7].

C-Flow seeks to provide the flexibility of hand-coding without the necessity of hand-written switch assembly code. Our implementation allows the programmer to write C code for each tile. The switch code necessary for communication between these tile programs is automatically generated by the C-Flow compiler. By generating switch code automatically, C-Flow simplifies the process of writing manually partitioned Raw programs. Thus, C-Flow's switch code generation facility allows the programmer to write larger programs more quickly, while maintaining performance similar to that of a program with hand-coded switch instructions.

The C-Flow programming interface has a certain similarity to the message passing interfaces frequently used in parallel and distributed computing. The MPI messaging interface [8] is frequently used in large cluster machines and is optimized for bulk data transfers. The single-word sized messages used in C-Flow would be very inefficient when used with such systems. Hardware message-passing machines, like the MIT Alewife system [9], are more capable of fine-grained messaging but still perform dynamic routing and scheduling, unlike C-Flow's statically scheduled messaging.

Chapter 2

An Overview of C-Flow

C-Flow allows the programmer to describe an application as a set of tile programs, each written in C. These tile programs, referred to as 'strands', can include communication operations that provide a basic form of message passing. Each strand can "send" or "receive" single data words to or from any other strand. The C-Flow compiler uses static analysis tools to automatically generate static network message schedules, unifying the separate "strands" into a single multi-tile "program". This chapter has two sections: the first describes the language interface used to access C-Flow from within C code and the second gives an overview of the C-Flow compiler system.

2.1 The C-Flow Programming Interface

C-Flow provides the application programmer with an MPI-like interface for sending messages between "strands". Each strand is composed of C-language code assigned to run on a particular tile or port. Strands can communicate with each other by sending messages across "channels". Channels are defined as global variables at the beginning of each C source file, and communication operations within function bodies send particular data words across these channels. This section defines the types of channels, the messaging operations, and the coding conventions that are used within C-Flow programs.

2.1.1 Endpoints and Groups

C-Flow allows two types of channels: endpoint channels and group channels. Endpoint channels are useful for point-to-point communication; they allow a strand to define some

Command	Argument	Argument Purpose
cf_endpoint	x	a tile's x coordinate
	y	a tile's y coordinate
cf_port_endpoint	x	a tile's x coordinate
	y	a tile's y coordinate
	dir	direction of port from tile ('N', 'S', 'W', 'E')
cf_rel_endpoint	x	offset in x direction from current tile
	y	offset in y direction from current tile

Table 2.1: Commands for Defining Communication Endpoints

distant target to which words of data may be sent. Group channels, on the other hand, define some set of strands which will perform a certain operation. For example, group channels are useful for broadcasts (one-to-many communication) and barriers.

While all channels are defined as global data at the beginning of a source file, there are several different methods for defining a channel. The mechanisms for defining endpoint channels are listed in Table 2.1. The basic *cf_endpoint(x, y)* form defines a channel between the current strand and the strand running on tile *(x, y)*.

Many C-Flow applications stream data on and off chip via Raw's external static network ports. In order to allow such communication, C-Flow allows the programmer to create "port" strands, pieces of code that generate the communication patterns that will occur on the static network port. Essentially, the programmer defines messages going to and from ports by creating a strand that would run on the opposite side of the port. Of course, when the final C-Flow compiled program is executed, there is no such tile on the opposite edge of the port, but the messages will go to and from the port as if there were. Consequently, when writing a "port strand", the programmer need only worry about the *order* of messages going to and from the port, not the *values* of the data being transmitted.

The other two endpoint definition mechanisms shown in Table 2.1 allow the programmer to define channels between tile strands and port strands. The *cf_port_endpoint(x, y, dir)* form defines a channel between the current strand and the "port strand" running on the specified port. The most general endpoint channel definition is *cf_rel_endpoint*, which defines a channel between the current strand and the strand located an offset of *(x, y)* from the current strand's location. If the offset defines a strand running off the edge of the grid, the channel is declared as communicating with the port adjacent to that off-chip location.

Group channels are defined by declaring each individual strand as a member of some

Command	Argument	Argument Purpose
cf_group_member	label	the name of the group containing this strand

Table 2.2: Commands for Defining Group Membership

group. The *cf_group_member("name")* macro, seen in Table 2.2, provides the mechanism for declaring group membership. The group "somename" is defined as the set of all strands which contain a *cf_group_member("somename")* declaration. A group may be composed of just two tiles, every strand in the program (ports and tiles), or any other arbitrary set of strands.

2.1.2 Communication Commands

Once a strand has declared communication channels, it can use C-Flow's messaging functions to communicate with other strands. Table 2.1.2 shows the various communication functions provided by C-Flow. Simple point-to-point communication is handled by the *cf_send(value, endpoint, "label")* and *value = cf_receive(endpoint, "label")* functions. C-Flow uses static analysis tools to match send and receive pairs on different strands while guaranteeing deadlock avoidance.

Group channels are used for two types of messaging: broadcasts and barriers. The *cf_broad_send(value, group, "label")* and *value = cf_broad_receive(group, "label")* functions are used to perform broadcast operations. The group specified in each of these commands is the set of all strands involved in the broadcast; i.e. the sending strand plus all receiving strands. The *cf_barrier(group, "label")* function provides a synchronization primitive; all strands in 'group' will wait at the barrier until every member of 'group' has arrived at the barrier.

All C-Flow communication commands take an optional "label" argument. This string provides a simple form of type-checking; when the C-Flow compiler's static analysis tools recognize a send/receive pair, they check that the labels match. This mechanism is useful for avoiding message ordering bugs; if tile 0 sends ("a", "b") but tile 1 receives ("b", "a"), a warning will be generated.

Command	Argument	Argument Purpose
cf_send	value	data word to be transmitted
	endpoint	strand that will receive the message
	label	a string identifying this message
cf_receive	endpoint	strand that sent the message
	label	a string identifying the message
cf_broad_send	value	data word to be transmitted
	group	set of strands that will receive the message
	label	a string identifying this message
cf_broad_receive	group	set of strands receiving broadcast
	label	a string identifying the message
cf_barrier	group	set of tiles to synchronize
	label	a string identifying this synchronization point

Table 2.3: Commands for Sending and Receiving Data

2.1.3 An Example Program: Chroma-Keying

In order to demonstrate the capabilities of C-Flow, this section provides a simple example program. This "chroma-keying" application accepts two incoming data streams from ports 0 and 15. The main strand, running on tile 0, examines the two input streams, and if a pixel in one of them is green, it is replaced with a pixel from the other stream. The result is then sent out of the chip via port 15. The end result is the green-screen effect used to place TV weathermen in front of maps.

Figure 2-1 show a graphical representation of the various strands involved in the chroma-keying application. The source files for each strand appear in Figures 2-2, 2-3, 2-4, and 2-5. The main program, and the only code that actually executes on Raw, is shown in Figure 2-2. This tile strand receives two input pixels, one from the camera and the other from a background image. If the camera pixel is green, the tile strand substitutes the background image pixel value and sends the result to the output port.

The astute observer will notice that this C-Flow application has three port strands but only two port locations: ports 0 and 15. This is possible because there is no control flow coupling between the incoming and outgoing buses on a particular port. Put another way, there is no switch processor on the opposite side of an external port, so there is no instruction stream governing when words cross the incoming and outgoing buses. Thus, C-Flow allows the assignment of Figure 2-4 to port 15's incoming bus and the assignment of Figure 2-5 to port 15's outgoing bus. Port 0, by contrast, has only one strand, shown in Figure 2-3. By allowing the programmer to assign separate strands to the incoming and

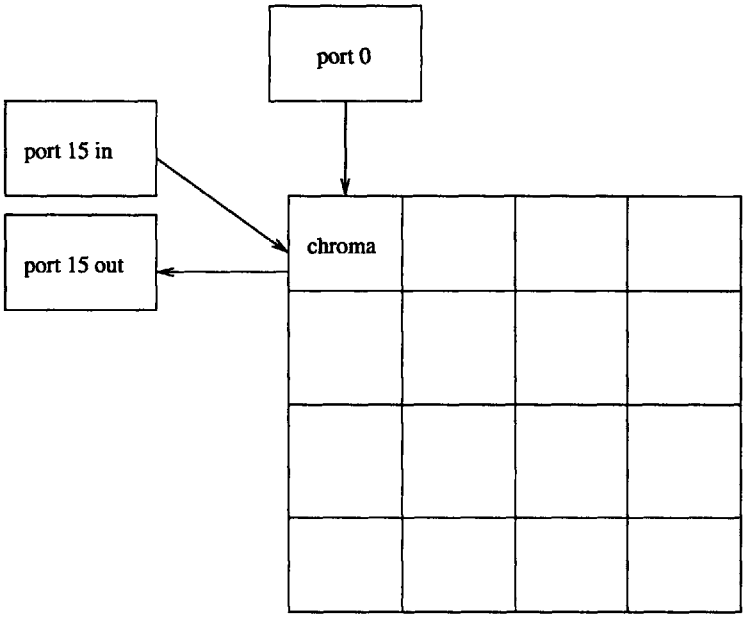


Figure 2-1: Chroma Keying Strand Assignment

outgoing buses on a particular port, C-Flow decouples incoming and outgoing messages. This decoupling may allow for more optimal scheduling of messages because, for instance, incoming messages need not necessarily wait for some outgoing result to be computed.

2.1.4 Matched Control Flow Depth

C-Flow uses static analysis tools to determine the ordering of communication operations between tiles. Given a set of strands as input, the C-Flow compiler uses basic-block analysis to extract the per-function control flow for each strand. In order to schedule a message between two strands, the C-Flow compiler must be able to guarantee that the receiving tile will be waiting when the sending tile sends its data word; if this is not the case, the system will deadlock as unreceived data words block up the static network.

Guaranteeing send/receive pairings requires that C-Flow be able to match up the control flow on the sending and receiving tiles. Our chroma-keying example satisfies this requirement. All of the input strand source files are composed of an infinite loop containing some messaging function calls. By examining the ordering of the sends and receives within the four main loop bodies, C-Flow will determine that each loop consists of three messages: (port 0, tile 0), (port 15, tile 0), and (tile 0, port 15). Thus, the number of loop iterations is the same for all communicating strands, and the ordering of communication commands

```
//chroma_tile.c

cf_endpoint port15 = cf_port_endpoint(0,0,'W');
cf_endpoint port0 = cf_port_endpoint(0,0,'N');

void begin()
{
  long blue, sub;
  long r,g,b;
  long result;                                                    10


  for(;;){
    blue = cf_receive(port15,"blue_in");
    sub = cf_receive(port0,"sub_in");

    b = blue & MASK;
    r = (blue >> RED_SHIFT) & MASK;
    g = (blue >> GREEN_SHIFT) & MASK;
                                                                  20
    if(((b > BLUE_LO) && (b < BLUE_HI)) &&
       ((r > RED_LO) && (r < RED_HI)) &&
       ((g > GREEN_LO) && (g < GREEN_HI)) ){
      result = sub;
    }
    else {
      result = blue;
    }

    cf_send(result, port0,"result");                              30
  }
}
```

Figure 2-2: Chroma main tile program

```
//port0.c

cf_endpoint tile0 = cf_tile_endpoint(0,0);

void begin()
{
  for(;;){
    int trash = 0xdeadbeef;

    cf_send(trash, tile0, "sub_in");
  }
}
```

Figure 2-3: Chroma data coming from port 0

```
//port15_send.c

cf_endpoint tile0 = cf_tile_endpoint(0,0);

void begin()
{
  for(;;){
    int trash = 0xcfcfcfcf;

    cf_send(trash, tile0, "blue_in");
  }
}
```

Figure 2-4: Chroma data coming from port 15

```
//port15_receive.c

cf_endpoint tile0 = cf_tile_endpoint(0,0);

void begin()
{
  for(;;){
    cf_receive(tile0,"result");
  }
}
```

Figure 2-5: Chroma results going to port 15

within the loop body matches up to form a sequence of send/receive pairs, so C-Flow can determine the overall message schedule and generate the corresponding switch code.

In order to match send and receive operations in two different strands, C-Flow requires that those two communication operations have the same control-flow nesting. That is, if strand A sends a word to strand B, and the "send" operation is inside of two nested loops, the "receive" operation in strand B must also be nested inside two loops. Thus, any control flow primitive (loop, if/else if/if block, switch statement) that surrounds a communication operation must appear on both strands involved in sending and receiving that message. Effectively, the control flow hierarchy must be the same in all communicating strands. Of course, control flow that is localized to only one tile, i.e. does not contain any sends or receives, need only appear on the tile that uses that control flow. Thus, in our chroma example, all of the strands must have their communication operations inside of the same infinite loop, but the main strand is the only strand with an "if/else if" block inside the body of that infinite loop. The C-Flow compiler requires that the communication operations appear in the body of the infinite loop, but it ignores any nested control flow that does not affect the communication schedule.

C-Flow's requirement of matching control depth means that some intuitively possible programs cannot be compiled by C-Flow. For example, Figure 2-6 shows a strand with two similar loops embedded in either half of an if/else block. Figure 2-7 shows a strand with a loop that, intuitively, would match up with either loop body. Essentially, the message schedule does not depend on the if/else block; regardless of the evaluation of the if statement, the message schedule will always consist of 10 messages from tile0 to tile1. However, C-Flow cannot compile and match up the message schedules in this program because it requires that the hierarchy of control flow be the same on both sides of a send/receive pair. Thus, in order to compile with C-Flow, this program would have to be changed so that tile 1 includes the if statement that exists on tile 0. The requirement of matching control flow hierarchy can be tricky, but in our experience it does not limit the types of algorithms that may be compiled with C-Flow.

2.1.5 Function Calls

C-Flow allows the programmer to make subroutine calls in order to encourage source code modularity. However, whenever a particular strand calls a function, C-Flow must guarantee

```
#include "raw.h"
#include "cflow.h"

tile1 = cf_tile_endpoint(0,1);

void begin()
{
  int num;
  int i;

  if(num < 5){
    for(i = 0; i < 10; i++)
      cf_send(i, tile1, "count");
  }
  else {
    for(i = 0; i < 10; i++)
      cf_send(10 - i, tile1, "count");
  }
}
```

Figure 2-6: Loops nested within an if statement

```
#include "raw.h"
#include "cflow.h"

tile0 = cf_tile_endpoint(0,0);

void begin()
{
  int num;
  int i;

  for(i = 0; i < 10; i++)
      num = cf_receive(tile0, "count");
}
```

Figure 2-7: A matching loop that will not merge

that all other strands that use that function make the call at the same time. If this guarantee failed, a tile could call a function, attempt to send a message, and hang because the message recipient had not made an equivalent function call.

In order to guarantee that functions are only called when all of their communication partners call the same function, C-Flow performs a series of tests at compile time. First, all functions with the same name are assigned to a "group function". Thus, if tiles 0, 1, and 2 all contain a function named "foo", the group function "foo" operates on the set (tile 0, tile 1, tile 2).

Having determined all of a program's "group functions", C-Flow checks to be sure that each group function satisfies several properties. First, if a group function 'A' is called by a group function 'B', the strands in 'A' must be a subset of those in 'B'. Secondly, C-Flow verifies that all communication operations within a group function operate only on strands that are member of that group function. Thus, a group function containing tiles 1, 2, and 3 may only communicate amongst those three tiles, it may not communicate with tile 4. Finally, when checking that the control flow hierarchy matches between strands, C-Flow verifies that any call to a group function happens at the same control flow point in all of the group function's strands. These verification steps will be explained in greater detail in Chapter 4, which describes the process of merging separate strands into a single unified program.

The only exception to the above rules is for functions from external libraries or functions which do not call any messaging commands. These functions will not send or receive any messages, so C-Flow can be assured that they will not result in a network deadlock condition. Allowing communication-free function calls without performing the group function checks is particularly important for library routines such as 'malloc'. Malloc() may be called on many tiles, and since it has the same name on all tiles, it would initially be marked as a group function. However, since malloc() does not use any communication operations, the group checks are not performed and strands may allocate memory independently of each other.

2.2 An Overview of the C-Flow Compiler

The C-Flow compilation infrastructure takes input in the form of C-language source code, analyzes the per-tile programs to determine messaging patterns, and generates the switch code necessary to schedule the program's messages. A graphical representation of the toolchain appears in Figure 2.2. The toolchain was designed to maximize code reuse; to that end, GCC and the Raw project's binary rewriting infrastructure play a significant role in compiling C-Flow programs. The next paragraphs will describe each of the compilation stages in detail.

A C-Flow application starts as a collection of C-language source files. Each source file may or may not contain some C-Flow directives. As specified in Section 2.1, there are two types of directives: channel definitions and communication commands. Channel declarations are used to assign a name to a particular communication route that will be used by communications commands.

The first stage of the compilation process uses the Raw project's GCC compiler to transform the C-language source code into a binary for each tile. The output binaries are instrumented with C-Flow commands and data so that later compilation stages can determine the messaging pattern between different tile programs. C-Flow channel declarations and communication commands are both inserted into the output binaries by a set of C preprocessor macros that are included in the C-Flow header file.

Channel declarations are stored as data words in the binaries' data segment; each word contains 4 bytes, indicating the x and y coordinates at the far end of the channel (the near end of the channel is implicitly the tile being compiled for), the type of the far end (tile or external port), and the off-chip routing direction for external port endpoints. Storing the channel information in the data segment allows the C-Flow macro routines to take advantage of gcc's C preprocessor. As a result, the arguments to a channel declaration may be either expressions or constants; this feature is particularly useful when using C macros to define several different strands within one source file. For example, certain channels could be declared only when the source file is compiled as a "port strand".

Communication commands are stored as 'move to static network port' instructions in the binaries' code segment. Each communication command in the code segment also has a corresponding annotation in a special "notes" segment added to the Raw binary format

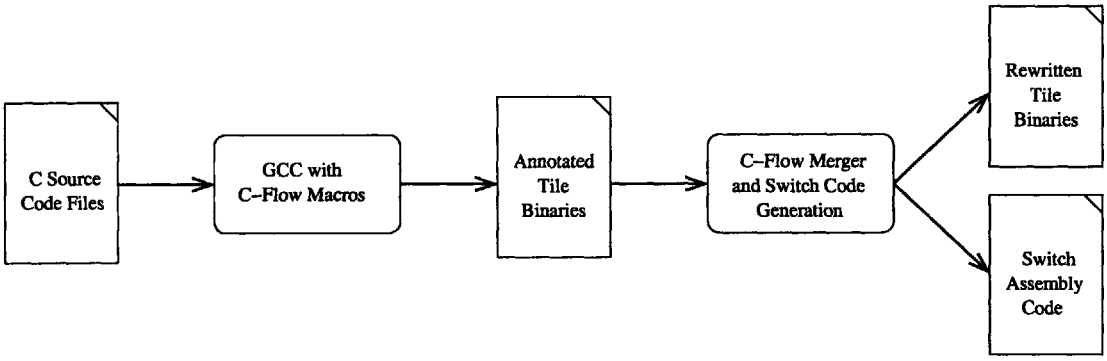


Figure 2-8: The C-Flow Compilation Toolchain

to help support C-Flow. By matching a communication instruction in the code segment with an annotation in the notes segment, later compilation stages can determine which communication channel is being used by each communication instruction. In addition to the communication channel name, annotations also include useful debugging information such as the command's source file line number and an optional name for each word sent.

All of the per-tile annotated binaries generated by GCC are used as the input to the next stage of the C-Flow toolchain, the merging and scheduling compiler. This stage in the toolchain has two phases. The first stage, described in Chapter 3, reconstructs the control-flow and message commands on a per-strand basis. The second stage, described in Chapter 4, merges the per-strand control flow into a single program, then schedules and routes each message operation. After both phases are complete, the output is stored as rewritten binaries for each tile and assembly code for each static switch. The rewritten binaries and switch assembly are then ready for assembly into a Raw boot image via the Raw project's standard assembler and linker tools.

2.3 Summary

This chapter has introduced the C-Flow language interface and compiler system. A C-Flow program is composed of several "strand" programs, each of which represent either computation and communication taking place on a tile or communication occuring across an external port. Each strand may contain two types of C-Flow primitives with its C-language code: communication operations and channel declarations. By matching the send and receive operations in different strands while guaranteeing that those strands contain the

same loop and conditional structures, the C-Flow compiler can determine a unique message schedule for the program as a whole. The next two chapters describe the process by which C-Flow rebuilds each strand's control flow and merges the strands into a single program.

Chapter 3

Extracting Control Flow from Object Files

As described in Section 2.2, each strand in a C-Flow program is passed to the compiler as an object file. The compiler is responsible for extracting the message schedule from the input strands and generating the switch code needed to execute that message schedule. The C-Flow compiler accomplishes this job in two phases. First, it converts the object file back into a structured program made up of loops, if/else statements, and compound conditionals. These control flow elements may be nested, forming a control flow hierarchy. The compiler takes each strand binary and extracts the control flow hierarchy that was originally in the C source for that strand. Once each strand's control flow hierarchy has been reconstructed, the compiler moves on to stage two, merging the separate strands into one program and generating switch code.

This chapter describes a method for rebuilding high-level control flow structures from object code. This process involves a number of graph transformations. First, basic blocks are extracted from a strand binary using the Raw binary rewriter infrastructure. Then, a control flow graph is constructed from the basic blocks and subroutines are identified. Each function is then processed to eliminate infinite loops and clarify control relationships, and finally a "program structure tree" (PST) [10] and "message structure tree" (MST) are created for each function in each strand.

The MST and PST describe each function in an object file as a hierarchy of single-entry, single-exit regions. Each region has an associated control flow type, for example loops and

if/else blocks. When nested within each other, these control flow region form a tree. The PST is a basic representation of this control flow hierarchy which shows only how the regions are nested within each other. The MST, derived from the PST, includes information on the order in which regions are executed and what messaging operations and function calls occur within each region. The C-Flow compiler's backend takes message structure trees as input; as described in Chapter 4, the backend merges the MSTs to obtain a global message schedule and generate switch code.

Throughout this chapter, the process of converting an object file into a program structure tree will be demonstrated via the example program in Figure 3-1. This program snippet performs no useful computation, but it does contain several interesting control flow structures. First, it contains a compound conditional, the 'or' statement with three terms. Then, the strand enters an infinite loop. The body of the loop contains an if/else if/else block. Together, these control flow structures constitute the majority of control flow blocks that exist within C programs. As this chapter describes the conversion from binary to program structure tree, each step will be demonstrated on this example program strand.

3.1 Extracting Basic Blocks from an Object File

The C-Flow compiler uses the Raw rewriter infrastructure, created by Jason Miller and Paul Johnson, to convert input binaries into an analyzable form. Using disassembly tools based on the GNU libbfd infrastructure, the rewriter transforms raw object code into a set of basic blocks. Basic blocks consist of a sequence of instructions that may only be entered at this first instruction and exited at the last. Branch or jump instructions may only occur as the last instruction in a basic block. Because basic blocks always finish by either falling to the next block, jumping to some other block, or branching, a basic block's exit point may lead to a maximum of two successor blocks.

After extracting basic blocks from the object file, the C-Flow compiler uses the rewriter to scan the binaries' ".note" and ".data" sections and locate C-Flow communication commands. As described in Section 2.2, channel declarations are stored in the strand's data segment and the routing information associated with messaging commands is stored in the note segment. Using this information, the compiler marks all basic blocks which perform network communication. It also links each communication instruction within a basic block

```
#include "raw.h"
#include "cflow.h"

tile1 = cf_tile_endpoint(0,1);

void begin()
{
  int random = 0;
  int num;
                                                                    10
  random = cf_receive(tile1, "random");

  if(random > 10 ||
     random < 0 ||
     random == 5){
    num = 1;
  }
  else {
    num = 0;
  }                                                                 20

  for(;;){
    if(num == 0){
      cf_send(1, tile1, "first");
    }
    else if(num % 2 == 1){
      cf_send(2, tile1, "odd");
    }
    else {
      cf_send(3, tile1, "even");                                    30
    }
  }
}
```

Figure 3-1: An example program containing several different control flow structures.

to the channel used by that instruction.

Once communication instructions have been identified and their channel usage noted, the compiler groups basic blocks into functions. Functions are identified as starting with some global label, such as "begin()" or "main()" and continuing until a return instruction is executed. Once all of a strand's subroutines have been identified, the basic blocks belonging to each function are identified and assembled into a final, per-function control flow graph. These control flow graphs contain all of the basic blocks in a function, the edges where control flow moves from one block to the next, and two special nodes: entry and exit. The entry and exit nodes denote the function's entry point and exit point, respectively, and their creation simplifies some of the later static analysis phases.

The input control flow graph for our example program appears in Figure 3-2. Every node in the graph, except for "entry" and "exit", represents some basic block, and the instructions within that basic block are shown inside the control flow node. Interestingly, the "exit" node is not connected to the rest of the control flow graph. This happens because our example program finishes in an infinite loop and does not exit. At this point in compilation, infinite loops are allowable, though, so this control flow graph can be passed to later stages in the C-Flow compiler.

In summary, the first phase of the control flow extraction process converts input object files into control flow graphs. Each strand's CFG is then analyzed to determine which blocks belong to which subroutines; the CFG is then split so that there is one control flow graph for each function in the binary. Finally, information from the .data and .note segments of the binary is used to annotate the per-function CFGs, indicating which instructions are messaging commands and what channel each message uses. At the end of this process, each strand object file has been converted into a set of control flow graphs, one for each function and each with its internal messaging commands flagged.

3.2 Preparing the Control Flow Graph

The remaining stages in the program structure tree process are designed to isolate single-entry, single-exit regions within each function's control flow graph. These regions can be characterized as loops, if statements, etc. in order to construct a tree representing the control flow hierarchy of different strands. Many of the algorithms used to perform this

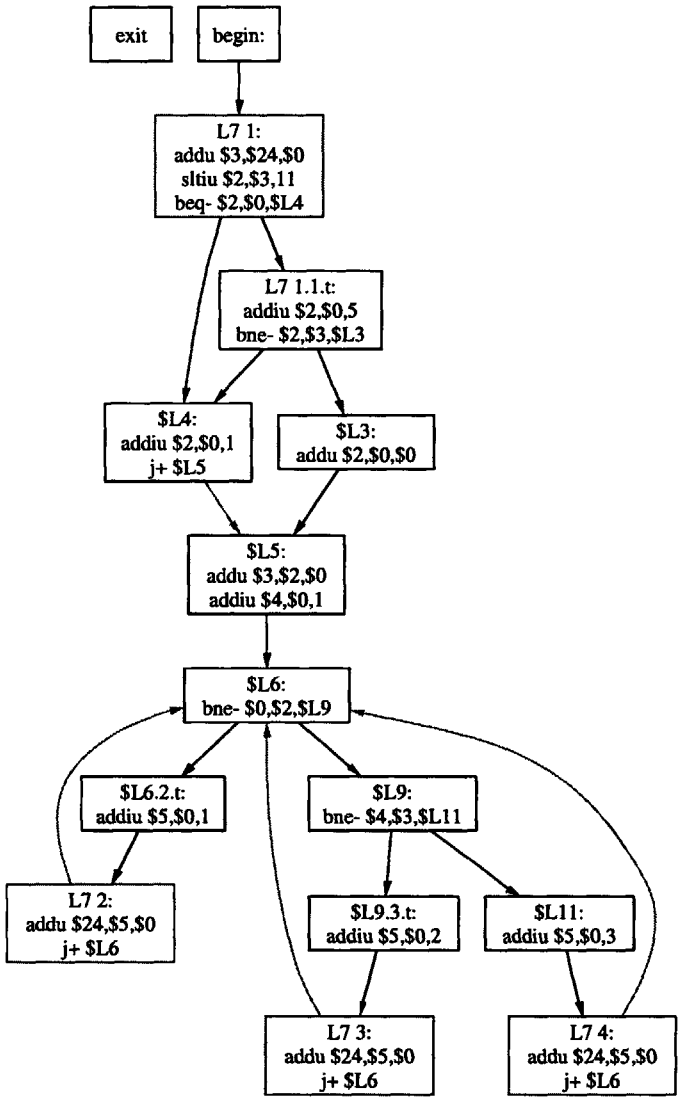


Figure 3-2: The initial control flow graph, with entry and exit nodes added

region identification require that the control flow graph have particular characteristics. For instance, our control graph already has the property that all basic blocks have no more than two exit edges.

The region recognition algorithms to be described in later sections required two other graph properties, as well. First, each function must have a single entry and a single exit node, and all nodes must be able to reach the exit node. This property forbids infinite loops, so infinite loops must be transformed into regular loops, as described in the next section. The infinite loop elimination algorithm and several later compiler stages also require that the graph be "complex-node free": any control flow graph nodes with multiple entry edges may have only one exit edge, and any node with multiple exit edges may have only one entry edge.

In order to establish this property, the C-Flow compiler splits any control flow nodes which have more than one entry edge and more than one exit edge. Each "complex node" is transformed into two "simple nodes". The first node has the original entry edges, the second node has the original exit edges, and a single edge goes from the first to the second. Thus, the overall control flow pattern is unchanged, but the graph now satisfies the "complex-node free" property. The results of this transformation on our example program appear in Figure 3-3; note that node $L6 from Figure 3-2 has been split into two nodes - "node_17" has all of the original node's entry edges, and the new $L6 has the original exit edges.

3.3 Eliminating Infinite Loops

Many compiler algorithms, including some in C-Flow, rely on "dominator analysis" [11]. A node 'a' dominates some node 'b' if all paths from the entry point to 'b' include 'a'. Thus, 'b' can only be reached by going through 'a'. Similarly, the node 'b' post-dominates 'a' if all paths from 'a' to the exit point contain 'b'. The 'dominator' and 'post-dominator' properties can be similarly defined for edges. The post-domination property is particularly useful for control flow analysis. For instance, if basic block 'a' has two outgoing edges due to a branch operation and basic block 'b' post-dominates 'a', 'b' will execute regardless of the branch condition in 'a'.

Unfortunately, many of the useful properties of post-dominator analysis are only useful in control flow graphs that do not contain infinite loops. The very definition of post-domination

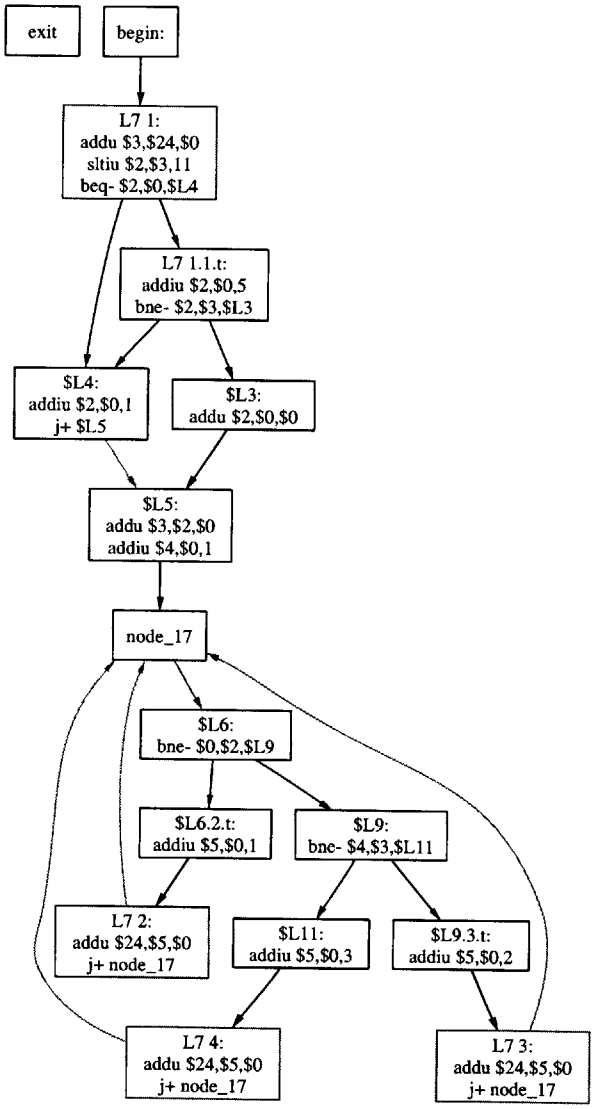


Figure 3-3: Control flow graph after node reduction

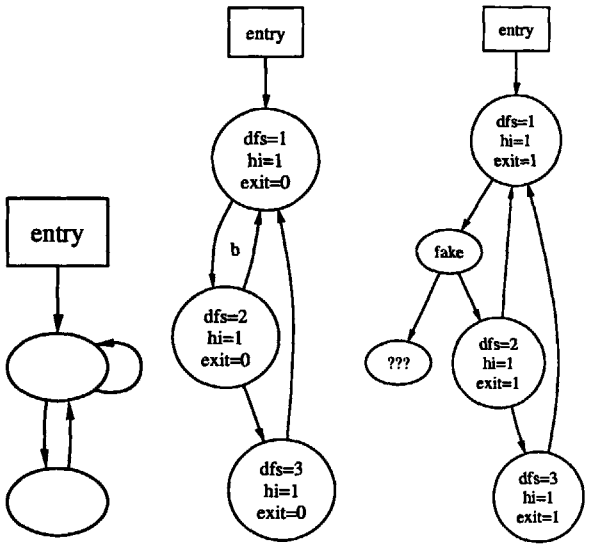
given above implies that all paths can reach the exit point, a prerequisite which fails if the CFG contains infinite loops. This section describes a method for removing infinite loops from a CFG by inserting "fake loop exits"; a branch within an infinite loop that appears to allow a path to the exit node even though the branch will never actually be taken. The process of removing infinite loops has two phases. After identifying infinite loops, we find a "fake exit point" or "breakout node" for each infinite loop. Then, a "reentry point" is found for each breakout, such that the reentry creates a path from the infinite loop to the exit node.

3.3.1 Identifying Infinite Loops and Finding Breakout Edges

Determining whether a control flow graph contains any infinite loops is straightforward. Each node is given a property *node.exit*, initialized to 0. Starting at the exit node, each node's predecessors are recursively marked with $node.exit = 1$, until all the nodes which can reach the exit node have been marked. If any nodes have $node.exit = 0$, an infinite loops is present. Discovering which nodes belong to which infinite loops is slightly more involved, but can still be done in O(N) time.

In order to identify which nodes are in which infinite loops, we add two properties to each node in the control graph: *node.dfs* and *node.hi*. *Node.dfs* is a node's index in a depth first traversal of the control flow graph, starting at the entry point. *Node.hi*, defined in [10], is the highest node in the depth first traversal tree that can be reached from a particular node (i.e. the minimum value of *node.dfs* that can be reached from a node). The value of *node.hi* can be determined in O(N) time using a reverse depth first order traversal of the graph in which *A.hi* is set to the minimum value of *node.hi* found amonst node A's children. That is, *A.hi* is set to the minimum value of *node.hi* that can be reached by following a single edge from A. An example control graph, including values of the above properties, appears in Figure 3-4(b).

An infinite loop exists if there is some non-exitable node ($node.exit = 0$) for which *node.hi* is less than *node.dfs*. Each unique infinite loop has a different value of "hi", and there is a unique node with dfs = hi. These nodes are referred to as "breakout nodes". Because each breakout node must have at least two entry edges, one entering the infinite loop and the other from the loop body, and the control flow graph has no complex nodes, we know that each breakout node will have a single exit edge. Any infinite loop that originally



(a) CFG from binary

(b) CFG after removing complex nodes

(c) Splitting breakout edge

Figure 3-4: Finding and splitting a breakout edges to remove infinite loops

did not have a single exit edge, as in Figure 3-4(a), will have one inserted when the complex nodes are eliminated as in Section 3.2. By splitting each "breakout edge" to insert a node with a "fake exit branch" leading to the rest of the program, the infinite loop is eliminated, as in Figure 3-4(c).

3.3.2 Finding Reentry Points for Breakouts

Once a "breakout" has been created for each infinite loop, the algorithm must determine where the breakout edge should reenter the control flow graph. An obvious choice is to have each breakout go directly to the exit node; this approach is simple and clearly guarantees that all nodes in the control flow graph will have a path to the exit node, as required for post-dominator analysis.

However, reentering directly at the exit node has some undesirable consequences. Consider the control flow graph in Figure 3-5(a). This program snippet consists of one if statement nested within another. One of the branches of the inner if statement leads to an infinite loop, which has already had its breakout point discovered. If the breakout is

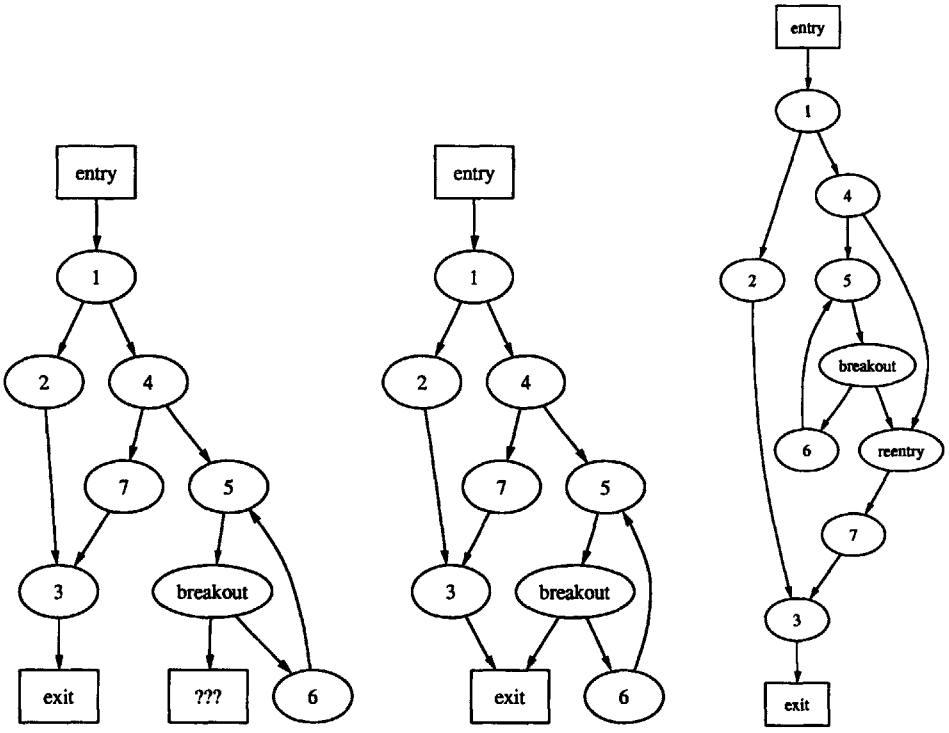
assigned to reenter at the exit node, we have a case as in Figure 3-5(b). In this case, some program structure has been lost because one of the branches of the inner if statement reenters after the merge point for the outer statement. Thus, what was once two nested if statements has turned into a valid control flow graph that could not be created in C without the use of 'goto' statements.

Since C-Flow relies on hierarchical control flow to merge separate strands, the loss of control flow structure in Figure 3-5(b) is problematic. Fortunately, the nested if statement structure can be preserved by selecting the reentry point as seen in Figure 3-5(c). In this case, the breakout edge has been given a reentry point in the other branch of the inner 'if' statement, preserving the nested control flow structure.

Fortunately, there is a relatively simple algorithm which selects the infinite loop reentry point so as to reenter the control flow graph without disturbing the nested control flow hierarchy. For each infinite loop breakout, we simply scan the CFG for a node which both dominates the breakout node and can reach the exit node. If such a node exists, we simply scan the path from the dominating node to the exit, searching for an edge which post-dominates the dominating node. When such an edge is found, it is split into two edges, a node inserted in the middle, and an edge added from the infinite loop breakout to the node in the middle of the split edge. If there is no node which both dominates the infinite loop breakout node and can reach the exit, the breakout node is simply attached to the exit node, as before.

For example, in Figure 3-5(a), nodes 5, 4, and 1 dominate the infinite loop. Scanning from the breakout node upwards, we find that node 5 cannot reach the exit node. Moving on, node 4 does have a path to the exit node, so we search for some edge on a path from node 4 to the exit which post-dominates node 4. Since node 4 has only one path to the exit, the edge between 4 and 7 immediately post-dominates 4. This edge is split, creating the reentry node, and the breakout edge is attached to the new node, as seen in Figure 3-5(c).

The example program from Figure 3-1 does contain an infinite loop. The control flow graph for this program, immediately before breaking the infinite loop, appears in Figure 3-3. The breakout node for this infinite loop is placed between "node_17" and "$L6". Nodes "$L5", "$L7_1", and "begin" all dominate the breakout, but none can reach the exit. Consequently, the infinite loop's fake reentry point is assigned to the exit node, as seen in Figure 3-6.



(a) CFG after adding breakout

(b) Reentry goes to exit

(c) Localized reentry

Figure 3-5: Determining where a breakout edge should reenter the control flow graph

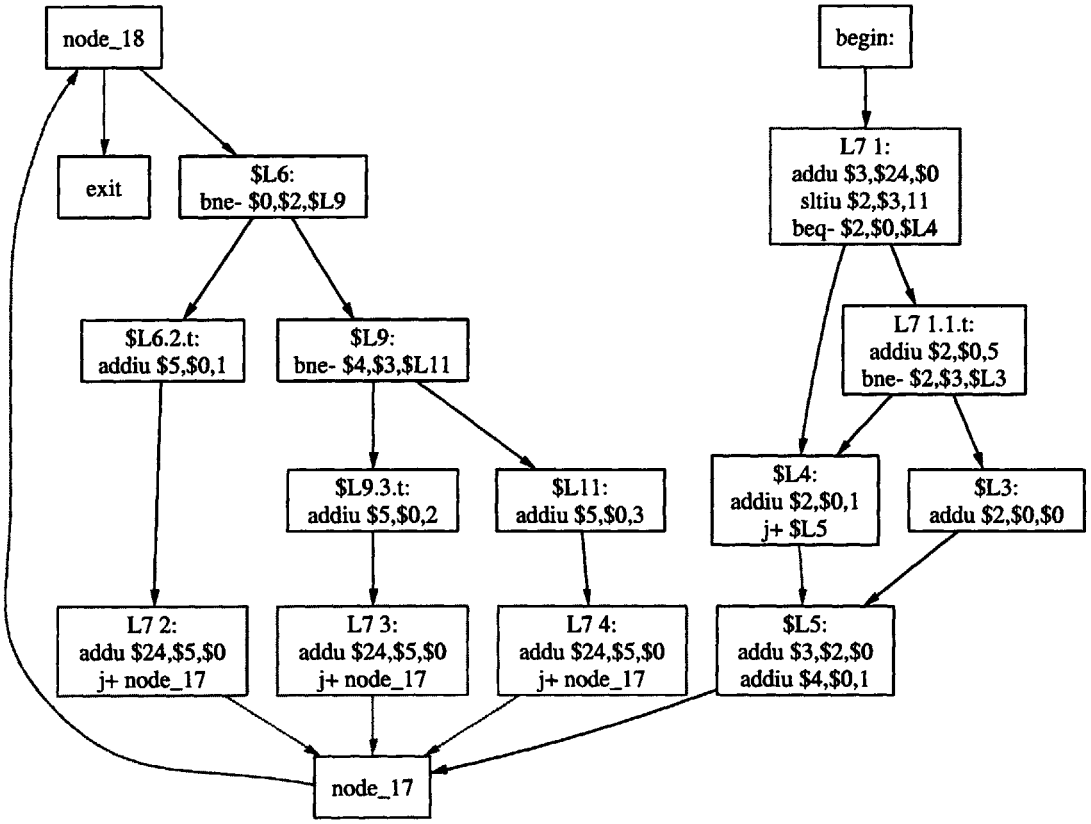


Figure 3-6: Control flow graph after eliminating infinite loops

3.4 Isolating Loop Bodies

The final transformation applied to each function's control flow graph before generating the program structure tree isolates a loop's body from its entry point. This separation is helpful when isolating control flow regions within the loop body; it allows the loop body to appear as a single-entry, single-exit region when the program structure tree is created.

Isolating the loop body from the entry requires identifying each loop entry node and determining which of the entry node's incoming edges come from the loop body and which come from the rest of the control flow graph (the entry edges). This information can be determined using dominator analysis, which is feasible now that all infinite loops have been removed. Using the dominance algorithm defined in [11], we first determine each control flow node's dominator set. Then, for each node 'a' in the graph, we check for any incoming edges coming from a node dominated by 'a'. If any such edges exist, the node is a loop entry point.

Once the loop entry points have been determined, the edges coming into each entry point 'a' are segregated into two sets, one composed of edges coming from nodes dominated by 'a' and the other composed of edges from nodes not dominated by 'a'. If there is more than one edge in either set, a new node is created to receive the edges in that set. A single edge is then created from the new node to 'a', thus creating a single edge by which all loop body nodes reach the entry point and a single edges by which all external nodes reach the entry point. The result of this operation on our example program appear in Figure 3-7. In this case, a new node "node_19" has been created. All of the edges from the loop body are connected to node_19, and a single edge goes from that node to the loop entry point, "node_17".

After isolating loop bodies, the control flow graph is ready to be converted into a program structure tree. At this point, the CFG has no complex nodes or infinite loops, its loop bodies have a single entry and a single exit edge, and each basic block is marked with the channel information for any messaging operations the block may contain.

3.5 Constructing the Program Structure Tree

A program structure tree (PST), first described in [10], provides a hierarchical description of a program's control flow structure. The nodes in a program structure tree represent

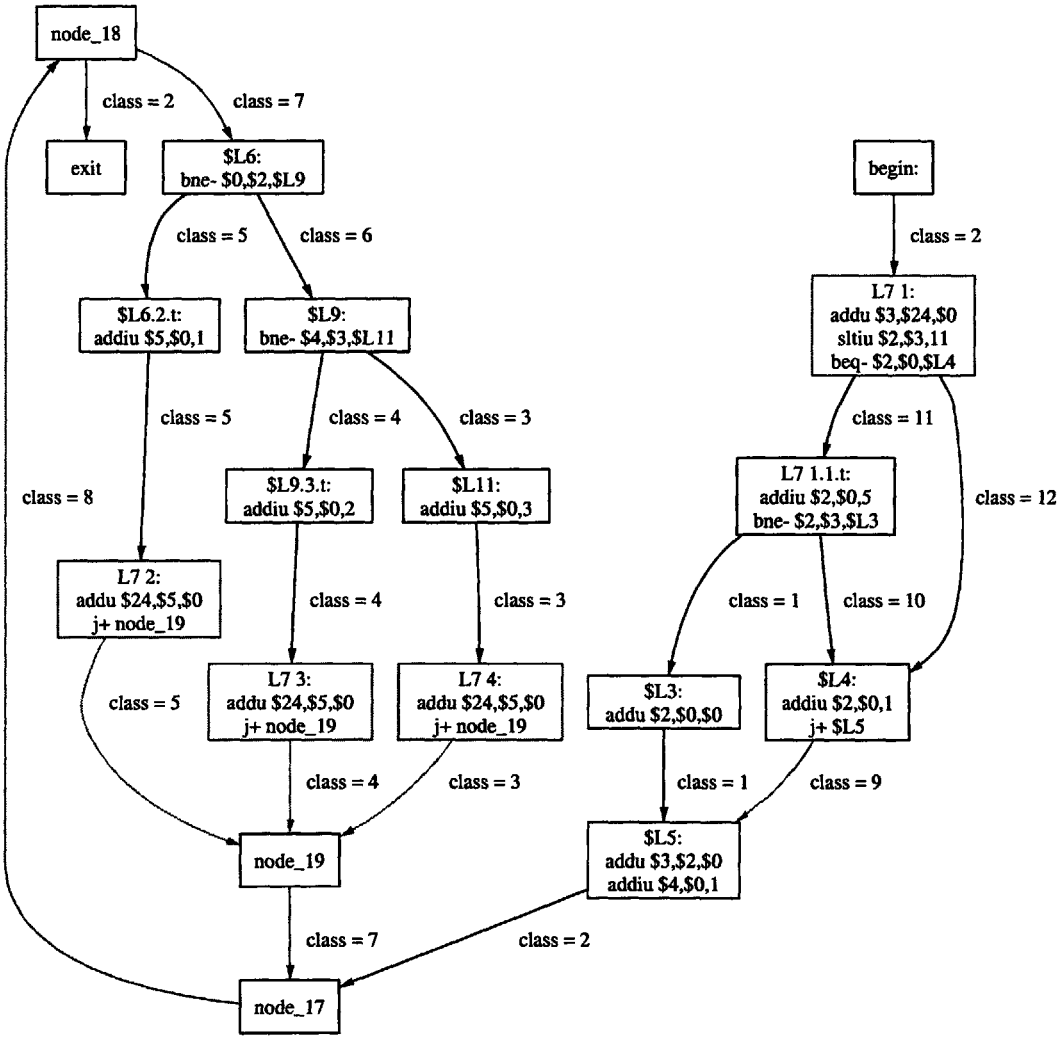


Figure 3-7: Control flow graph after isolating loop bodies

single-entry, single-exit (SESE) regions within the control flow graph. These SESE regions may be nested within each other to form a tree. For example, an if statement has a single entry point (before the branch) and a single exit point (after the two forks merge), so an if statement will appears as a SESE region in the control flow graph. An if statement might be embedded within a loop body, which is inside a simple loop (another SESE region), which itself might be inside some other control flow structure. Thus, the program structure tree represents the nesting of SESE regions in the CFG. Within the PST itself, nodes represent SESE regions and edges represent the nesting of one region within another. Thus, the program structure tree is simply a heirarchical representation of the control flow graph; the PST can be built from a CFG and vice versa.

For example, Figure 3-8 shows the program structure tree corresponding to the control flow graph after loop body isolation (Figure 3-7). The top level function, begin(), has four sequential SESE regions. A careful comparison of this PST to the control flow graph it represents reveals that each SESE region corresponds to some element of the hierarchical control flow. For example, the SESE region that begins with "L7_1" and ends with "$L5" corresponds to an "or" statement in the original program.

The creation of the PST is performed using the algorithm described in [10]. This algorithm runs in O(N) time and is based on the concept of "cycle equivalence". Two edges in the control flow graph are cycle equivalent iff for all cycles C, C contains either both edges or neither edge [10]. It turns out that marking each edge with a "cycle equivalence class" makes the recognition of SESE regions trivial - any pair of edges in the same class forms a SESE region. The control flow graph in Figure 3-7 has the cycle equivalence class number marked on each edge. The relationship between cycle equivalence and the SESE regions in the program structure tree is apparent; for example, top level SESE regions in the program structure tree are bordered by edges in cycle equivalence class 2.

3.6 Region Type Recognition - the Message Structure Tree

Having created a program structure tree for each subroutine in each strand object file, the C-Flow compiler is almost ready to begin merging the separate strands into a single global messaging schedule. However, two characteristics of the PST make it less than ideal for use in merging. First, the leaf nodes in a PST are basic blocks, but message schedules are

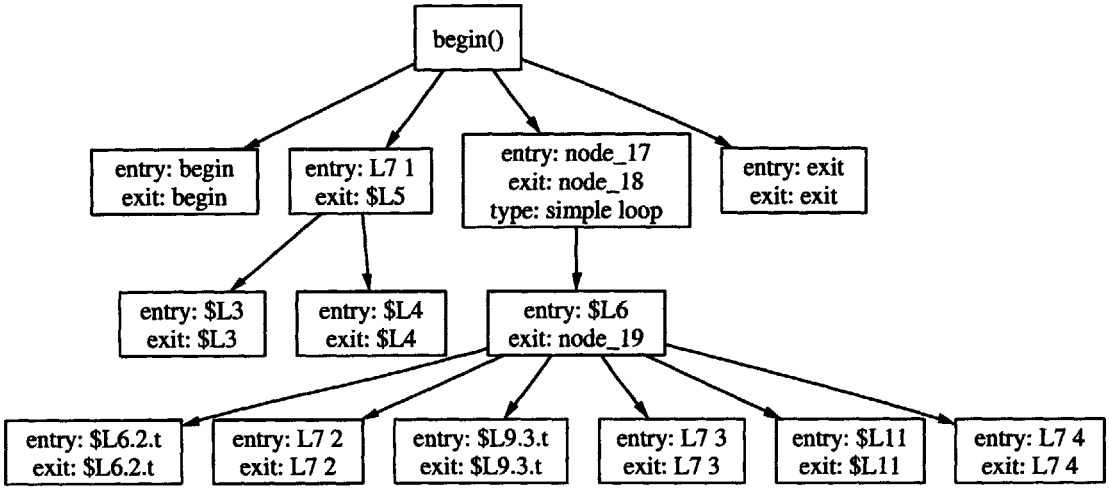


Figure 3-8: Program structure tree for our example program

a function of messaging commands within those basic blocks. Secondly, the single-entry, single-exit regions within the PST do not have any type information. That is, the PST recognizes SESE regions and describes their nesting, but it does not indicate whether a particular region is a simple basic block, an if statement, or a loop. Having this type information is critical in order to guarantee that send / receive pairs in different strands have the same control dependencies.

In order to address these weaknesses, we introduce the "message structure tree" (MST). Like the PST, the message structure tree has nodes representing SESE regions (with added type information), but it allows several other node types as well. These other nodes include sequences, function calls, and messaging operations. Function calls and messaging operations are always leaf nodes in the MST; they are the primitives contained in the bodies of loops and other control flow structures. The body of the MST is made up of alternating layers of region nodes and sequence nodes. The rules governing which node types are allowed as children of a particular node type are summarized in Table 3.6; the purpose of each node type and the reasons for these rules will be explained in the next few sections. For reference, a MST corresponding to our example program appears in Figure 3-11.

3.6.1 Sequences

Sequence nodes in the MST provide an ordering for their child nodes. For example, the node "entry seq" in Figure 3-11 has three children, indicating the program will first receive

Node type	Child types
Sequence	SESE region, function call, messaging
SESE region	sequence
function call	none
message	none

Table 3.1: Commands for Defining Communication Endpoints

a value "random" from tile.0.1, then perform an "or-type" conditional, then enter a loop. SESE region nodes always have sequence nodes as children. Some regions, such as the function's entry point, have a single child execution sequence; others, like if statements, have two child sequences. The process used to determine the order of a sequence's child nodes will be explained in Section 3.6.3.

3.6.2 SESE Control Flow Regions

The C-Flow compiler can recognize four types of SESE regions within the MST: function entry points, if statements, compound conditionals (or), and simple loops. The root node of a message structure tree is always an "entry point" region; this type of node has exactly one child node, a sequence node that represents the execution of all the top level control-flow within the function. The other three region types correspond to some subset of the nodes in the function's control flow graph. Regions in the PST keep a record of their entry and exit edges within the CFG; when PST regions are converted into MST regions, the control flow graph around the entry and exit edges is examined in order to determine the region's type. The templates for each type of region (except the function entry point) appear in Figure 3-9.

Within the control flow graph, each single-entry, single-exit region type is composed of some pattern of entry and exit nodes with sequences of other SESE regions between. The location of these "embedded sequences" within each SESE region type is indicated by a box in the templates shown in Figure 3-9. When the MST is constructed, each SESE region type associates a particular type with its child sequences. For instance, if statements have two child sequences, one to be executed if the branch is taken, the other taken if the branch falls through.

Each of the three region-type templates serves a particular purpose. The "loop" region, seen in Figure 3-9(c), occurs whenever the original C code has a for(), while(), or do-while()

loop. A "loop" region has two child sequences. In the control flow graph, the first sequence is executed between the region's entry node and the branch operation that exits the loops. The second executes after the branch operation and includes a jump back to the loop's entry node. In the MST, these child sequences are referred to as the "pre-branch" and "post-branch" sequences, respectively.

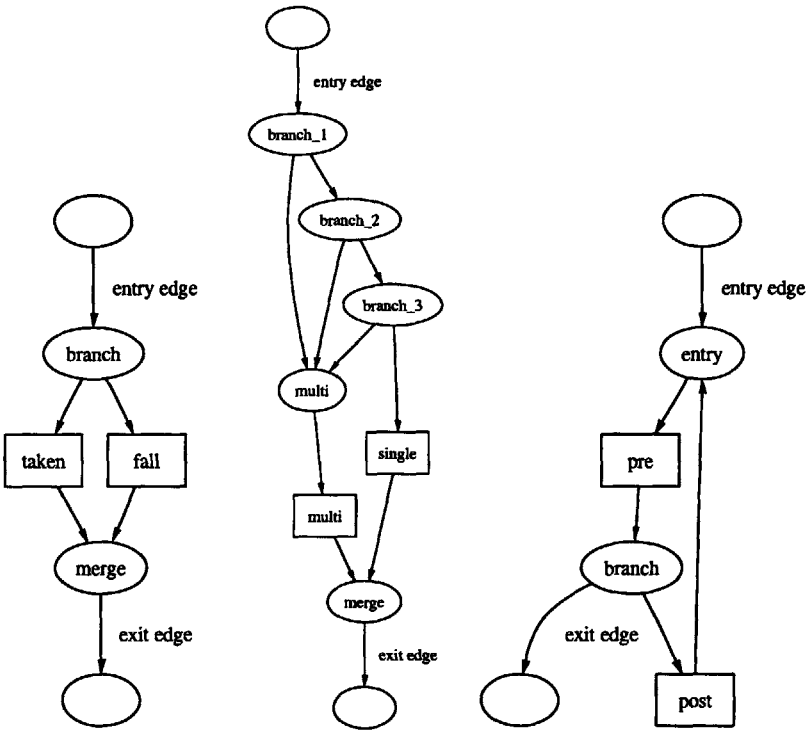
"If" and "or" regions occur as a result of if() statements in the source code. Both of these region types have two child sequences, one to be executed if some conditional is true, the other to be executed if it is false. The "if" region, shown in Figure 3-9(a) represents the simple case, with a conditional represented by a single branch operation. An "if" type region has two child sequences; the "taken" sequence is executed if the branch is taken, and the "fall" sequence is executed if the branch falls through.

The "or" type region in Figure 3-9(b) occurs as a result of more complex conditionals in the original source code. Like an "if" type region, the "or" type has two child sequences which merge before the exit edge. However, the control flow nodes following the entry node may be more complex; multiple branch operations may be needed to determine which child sequence is executed. The entry to an "or" type region is composed of a series of branching blocks. One outgoing edge of each branch operation goes to the next branch operation, and the other to the "multi" sequence". The final branch operation has one edge leading to the "multi" sequence and the other leading to the "single" . Thus, if any of the branch conditions is met (i.e. (branch 1) or (branch 2) or ...) the multi-sequence is executed; otherwise the single-sequence is executed.

3.6.3 Constructing the Message Structure Tree

The message structure tree is constructed by copying the program structure tree while referencing the control flow graph to determine SESE region types. Once a region's type is determined, the control flow graph is also used to insert messaging operations and function calls within its child sequences. The copying process can be performed as a pre-order depth first traversal of the PST, performing three operations at each node.

Each node in the PST being converted into a MST represents a single-entry, single-exit region within the control flow graph. The first step in converting each node is to determine its type by comparing its entry and exit node patterns to the templates in Figure 3-9. If the region is an "if" or an "or" type region, it may require some extra processing to



(a) if (b) or (c) loop

Figure 3-9: Single-entry, single-exit region types identified by C-Flow. Boxes represent a "sequence", which may be composed of messaging commands, function calls, or other single-entry, single-exit region types.

determine its child sequences. Whereas loop bodies are guaranteed to be single-entry, single-exit regions, the two alternative execution paths in a conditional may be single-entry, multiple-exit regions, which will not appear in the PST.

For example, consider the nested if statement in our example program. Examining the control flow graph in Figure 3-7, we see that node "$L6" should be recognized as the entry to an "if-type region". Put another way, the PST node in Figure 3-8 with entry = $L6 and exit = node_19 should be an "if" type region. However, the right-hand side of this if statement, starting with node "$L9", will not be recognized as a sequence of SESE regions because it has multiple exit edges going to node_19.

Thankfully, such single-entry, multiple-exit regions are easily fixed. Starting at the entry edge to each of the sequences, we first run a depth first search to guarantee that the two sequences do not have any common nodes. If this condition is met, we create extra nodes to group the exit edges from each sequence separately. This process creates one or two new child SESE regions in the control flow graph, which are then added to the PST. Adding nodes to the PST is permissable because the tree is traversed in pre-order so that the new child nodes are guaranteed to be copied later. The result of this transformation on our example control flow graph appears in Figure 3-10

Once all possible child SESE regions have been identified, the final two steps in the per-node copying procedure may be performed. First, the child regions in the PST are grouped in sequences based on the "cycle equivalence class" of their entry and exit edges. Regions in the same sequence are guaranteed to have the same edge class and may be ordered according to their common edges. Once the child sequences have been constructed, they are associated with their parent region. Finally, any messaging operations or function calls with the current SESE region but not in any child regions are inserted into an appropriate sequence node.

The final message structure tree for our example program appears in Figure 3-11. It closely resembles the program structure tree from which is was derived (Figure 3-8), but includes region types and has each region's child regions sorted into sequences. Also, the MST includes message operations like send and receive as leaf nodes. This message structure tree is now ready to be merged with other strands in the program in order to generate a global message schedule; this process will be explained in the next chapter.

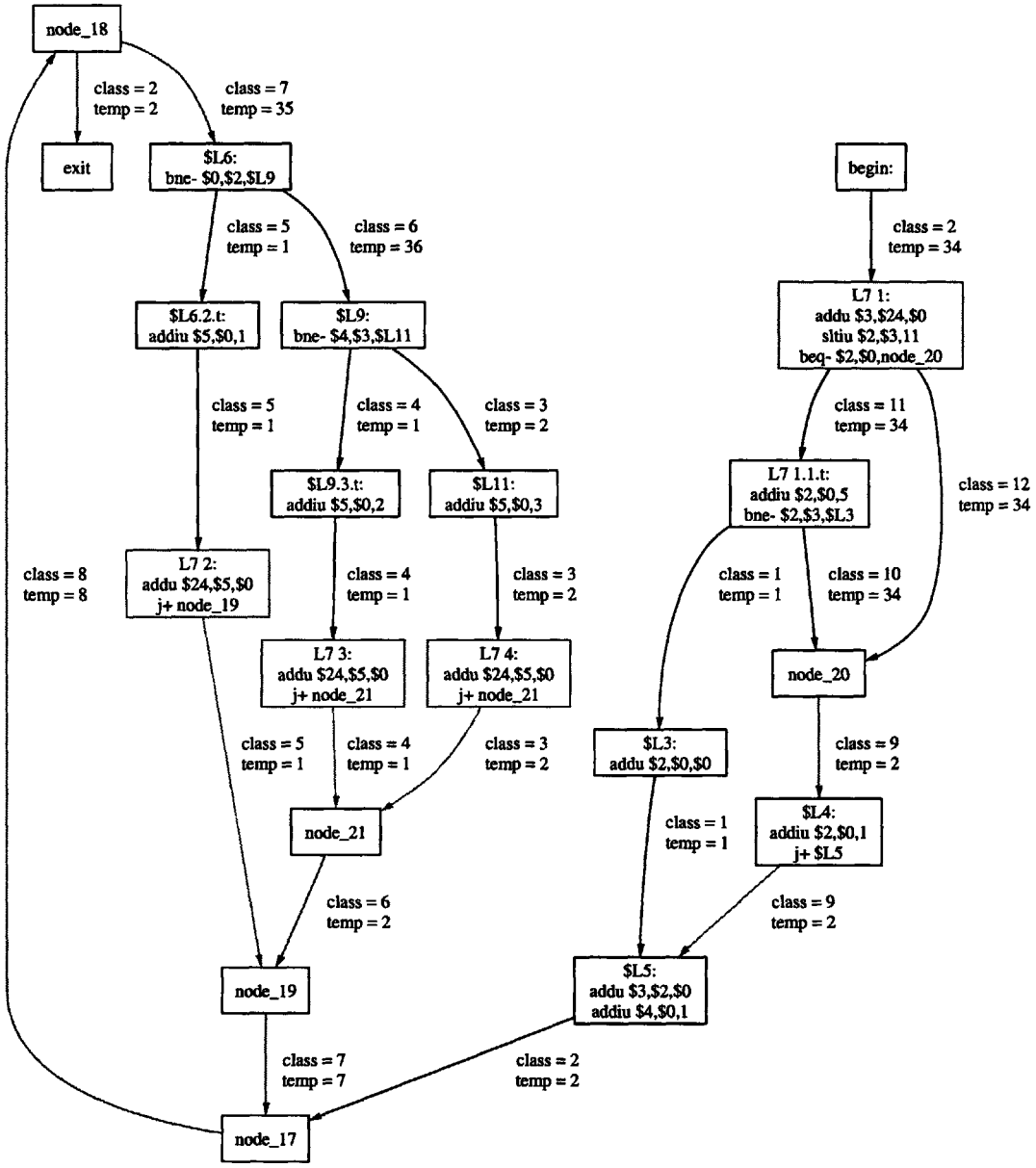


Figure 3-10: Control flow graph after region type identification has transformed single-entry, multiple-exit regions into single-entry, single-exit regions

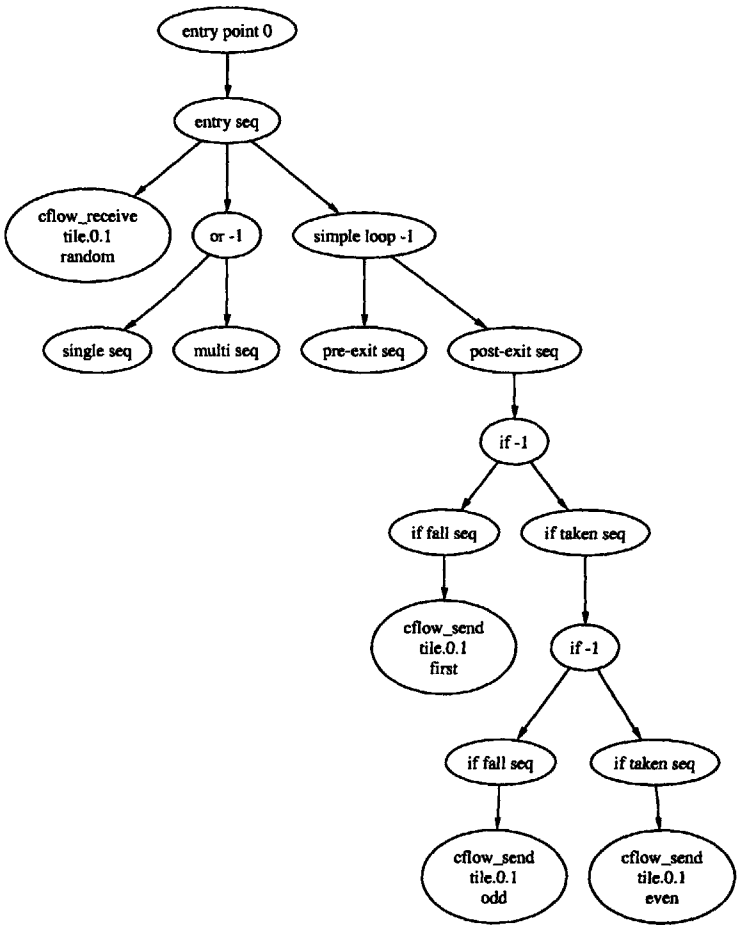


Figure 3-11: The message structure tree derived from our example program

Chapter 4

Merging Control Flow and Scheduling Messages

The C-Flow compiler infrastructure uses the gcc compiler as a "frontend" and then uses the binary rewriting techniques described in Chapter 3 to construct a "message structure tree" for each strand in a program. Once a MST has been prepared for every tile and port in the program, the C-Flow compiler generates a "unified message tree" (UMT), which provides a unified description of the messages and control flow contained in the program. The UMT is similar to a MST or PST in that it gives a hierarchical description of the program's nested control flow, but it unifies the input message structure trees so that one UMT represents many MSTs worth of data.

The UMT serves several purposes. First, our algorithm for UMT construction guarantees that, if the UMT can be built, the program is deadlock free. If UMT construction fails, the algorithm can also provide a specific list of which messaging operations led to a cycle in the program's communication pattern. Secondly, the UMT allows analysis of different control regions in the program - for example, half the tiles in a program could be running a synchronized loop while the other tiles performed their own individual work, only to be resynchronized later. Finally, the UMT provides an easy program representation for routing and scheduling messages.

This chapter contains three sections. The first introduces the unified message tree, giving several example and describing important details of this intermediate representation. The second section gives an algorithm for building a UMT while guaranteeing that the

resulting program is deadlock free. Finally, the third section gives an overview of how to generate switch code using the UMT, routing messages and copying control flow from the tile processor to the switch as needed.

The algorithms presented in this chapter are designed to generate simple, deadlock-free switch code. Many performance optimization are possible within this framework; these will be described in Chapter 5.

4.1 The Unified Message Tree

The unified message tree can be thought of as a "merging" or "superposition" of a program's many message structure trees. As in a MST, the UMT's internal nodes represent control flow operations (loops, ifs, ors) and sequences of events. The leaf nodes represent message operations and function calls. However, the MST's nested control flow nodes include a list of which strands execute that control flow, and the leaf nodes represent all the strands involved in sending or receiving a particular message.

For example, consider the chroma-keying application first shown in Section 2.1.3 and Figures 2-1 through 2-5. The message structure trees for each input strand (tile 0, port 0, port 15 send, and port 15 receive) appear in Figure 4-1. The unified message tree built from these four input strands appears in Figure 4-2.

The chroma-keying UMT provides some very useful information. Reading from the root of the tree, the entry point node, the UMT indicates that the "begin" function consists of a single loop. The body of that loop first checks the loop exit condition and then executes three messaging operations. The first two route the input data from ports 0 and 15 to tile 0, and the third routes the result from tile 0 back out port 15. Between the input and output messages, tile 0 also executes some nested if statements, which are actually used to perform range checks on each of the three input color channels. These nested if statements are not executed on any of the port strands.

The following subsections will give a more detailed description of the information provided by a UMT. Most importantly, the UMT provides a deadlock-free ordering for all of a program's control flow and messaging. Secondly, its internal nodes may actually represent control flow on a subset of the input strands, so that tiles performing computation without messaging are not unnecessarily synchronized with other tiles. After giving a detailed

description of the "ordering" and "control flow subset" properties, we proceed to a more involved example program. This second example is specially designed to accentuate several difficulties in the UMT construction algorithm given in Section 4.2.

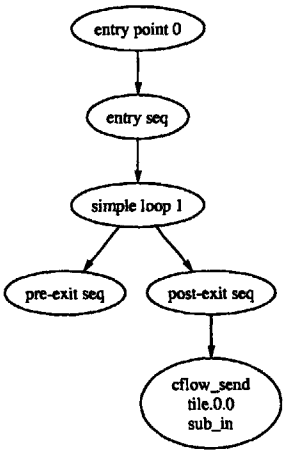
4.1.1 Ordering in Unified Message Trees

Like message structure trees, unified message trees contains "sequence nodes" which provide an ordering for their child nodes. Sequence nodes may only occur at odd-number depths in the UMT because control flow nodes have sequences as children and sequences may not have other sequences as children. Thus, the root node in the UMT is a control flow node (the entry point), its child is a sequence, that sequences has other control flow nodes as children, etc. Each sequence node provides a total ordering for its child nodes. For example, node "post-exit seq" in Figure 4-2 shows that the program will transmit a message named "blue_in", then one called "sub_in", then the tile will execute some nested ifs, and finally a message named "result" will be sent.

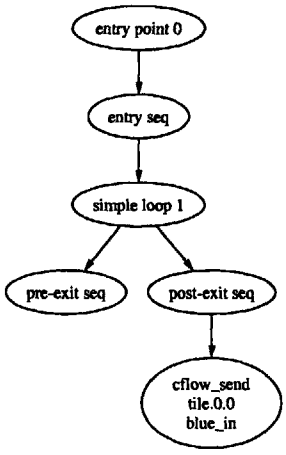
The ordering property is critical for deadlock avoidance. If a tile expects to receive "sub_in" and then send "result", the UMT must guarantee that ordering is maintained in the global messaging schedule. Of course, there may be more than one deadlock-free message ordering; our basic UMT construction algorithm simply picks one and guarantees that it is deadlock-free. More advanced schedulers could more optimally order the messages and nested control flow in a sequence in order to improve performance.

Because sequence nodes are guaranteed to alternate layers in the UMT and MST, they also provide a total ordering for the entire program. Each control flow node provides some implicit ordering of its child sequences - for example, an "if" region executes one child sequence or the other. Consequently, a pre-order traversal of a UMT or MST provides a total ordering of the messages, function calls, and control flow operations in that function. For example, the chroma-keying UMT can be ordered as follows:

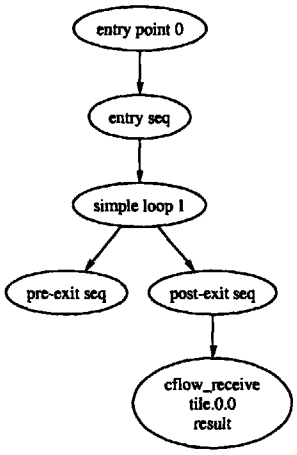
1. Entry point: function is entered on all four strands
2. Entry seq:
3. Simple loop: entered by all four strands
4. Pre-exit sequence



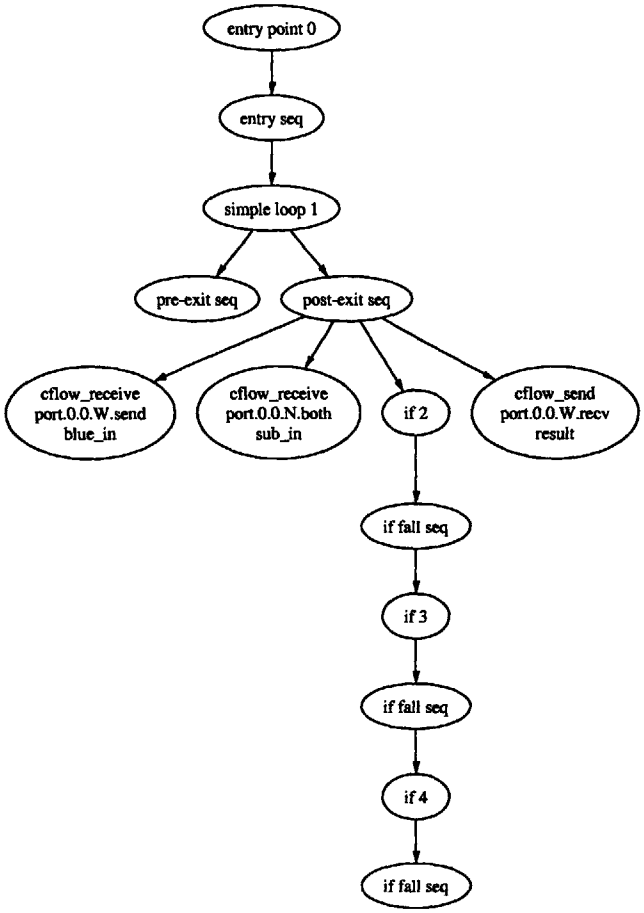
(a) port0



(b) port 15 send



(c) port 15 recv



(d) tile 0

Figure 4-1: The message structure trees for each of the four strands in the chroma-keying application.

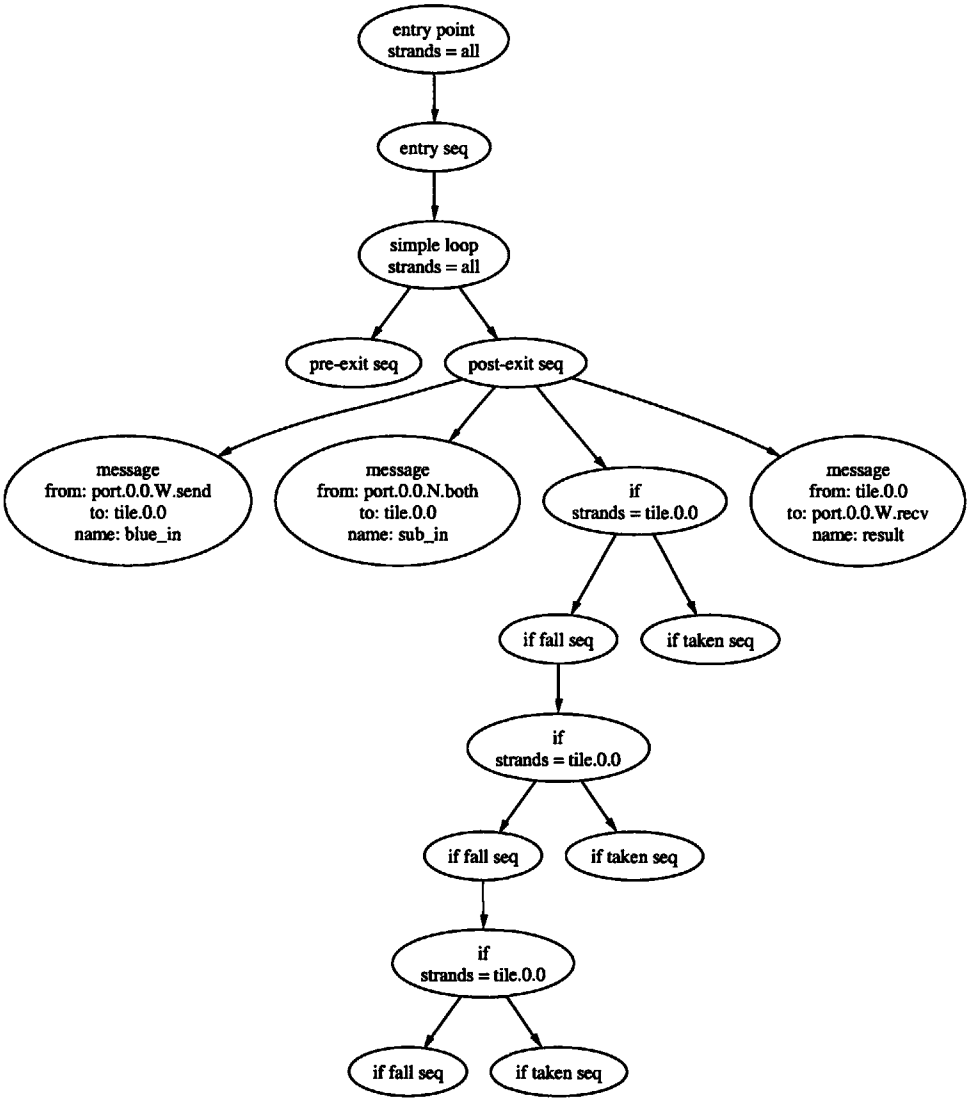


Figure 4-2: The unified message tree for the chroma-keying application

5. Post-exit sequence

6. Message: blue_in

7. Message: sub_in

8. If: only on tile 0

9. If fall seq: if the branch op falls through, ...

10. ... two more nested ifs ...

11. Message: result

12. Items 4 through 11 are repeated some number of iterations...

This total ordering property of the UMT as a whole guarantees deadlock avoidance: if each sequence's children are deadlock free, and each control flow operation selects the same set and ordering of its child sequences, the entire tree will be traversed in some defined order. Since this ordering is the same on all tiles, the message-ordering will have no cycles and the program is deadlock free.

4.1.2 UMT Nodes and Control Flow Subsets

Unified message trees also keep track of which strands execute a given piece of control flow. For example, the chroma-keying example in Figure 4-2 executes an infinite loop in all strands, but only the "tile.0.0" strand performs the "if regions" needed to check whether each incoming pixel is green or not. Within the UMT, this information is encoded in a "strands" data member found in each control flow node. Thus, the "simple loop" node is executed by all four strands, but each of the "if" nodes are executed only by tile.0.0 (tile 0). The set of strands that actually execute a particular control flow element is referred to as its "strand subset".

The "strand subset", referred to as "strands" in Figure 4-2, must contain at least the strands which are needed by any leaf nodes descendant from the control flow node of interest. That is, if a control flow node has a child sequence which sends messages between tiles 1, 2, and 3, then that control flow node's "strands" member must contain at least those tiles. A larger set of tiles may be useful if extra tiles are needed for routing purposes; for example,

if a control flow node has a child message between tiles 0 and 2, it must also include tile 1 so that the switch on tile 1 will be at the appropriate control flow point to route a message word between tiles 0 and 1. In this case, the necessary branch conditions will have to be copied to tile 1 from tile 0 or 2 via the static network.

Our UMT construction algorithm builds control flow nodes with the minimal subset of strands, ignoring the above routing consideration. This minimal subset is useful because it minimizes the control flow synchronization between tiles. Thus, tiles 0 and 1 could execute a loop at the same time as tiles 2 and 3 execute a different loop, and there need not be any synchronization between the two loops. This freedom from unnecessary synchronization benefits performance by avoiding extra synchronization overhead and unnecessary serialization, thereby increasing potential parallelism.

4.1.3 A More Difficult Example

The process for merging the chroma-keying MSTs into a UMT seems relatively straightforward. There is only one loop, and all of the strands are involved in that loop, so matching the control flow in the different MSTs is intuitive. Additionally, since all the messages in that loop body involved the strand running on tile 0, there is only one valid deadlock-free message ordering - the order in which messages are processed on tile 0.

Generating a UMT for more complicated programs can be more difficult, particularly given the requirements of deadlock-freedom and minimal strand subsets. For example, Figure 4-3 shows the input MSTs from a program that runs on tiles 0 through 3. The UMT that results from the merging of these four MSTs appears in Figure 4-4.

This example contains several new complications. First, the application includes a broadcast from tile 1 to tiles 2 and 3. In order to avoid deadlock, the UMT construction algorithm must guarantee that the latter two tiles will be waiting for the broadcast when tile 1 sends it. Secondly, the program finishes with "if regions" on three tiles. Two of the tiles, tiles 2 and 3, call a function from within the if statement. The other, tile 1, does not. According to our minimal strand subset rule, only tiles 2 and 3 need be in the same "merged" if block. As a result, the UMT contains two independent "if regions"; one executes on tiles 2 and 3, the other on tile 1. Consequently, the two tiles are guaranteed to make their function call at the same time, avoiding deadlock, but the third tile is not unnecessarily synchronized to that function call.

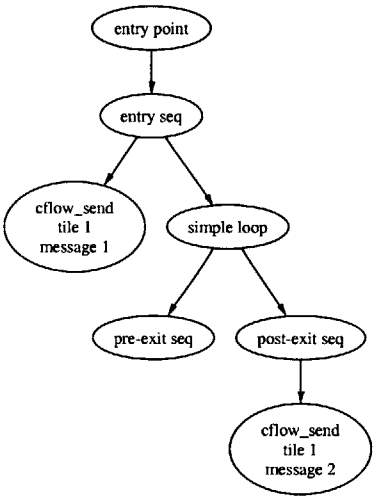
Additionally, note the ordering of the messages within the loop body. "Message 2" and "message 3" involve disparate sets of tiles, so the two messages could be transmitted in reversed order and still avoid deadlock. "Message 4", however, must come after these two because it involves operations on tiles 2 and 3 which must happen after the previous messages. These ordering constraints are apparent from the input MSTs. Thus, we see that there is more than one valid deadlock-free ordering, but that care must be taken to put "message 4" after the others.

As a final note, this application also motivates the use of a "match list" in the UMT construction algorithm that will be described in Section 4.2. While this characteristic will make more sense once the algorithm is presented, a brief note at this point may justify the creation of the "match list" later. Notice again that "message 2" and "message 3" involve different sets of tiles. If these were the only two messages in the loop, the minimum strand subset rule would require that the UMT have two loops - one for tiles 0 and 1, the other for tiles 2 and 3. However, if a third message is added using tiles from both subsets, as in "message 4", the two previously separated loop bodies must become one. As the construction algorithm progresses, it initially constructs the two separate loops because it has only seen the first two messages. When the third message is found, the two separate loops must become one. The "match list" is used to accomplish this task. Again, the algorithm is presented in depth in the next section; here we are only motivating an aspect of that algorithm.

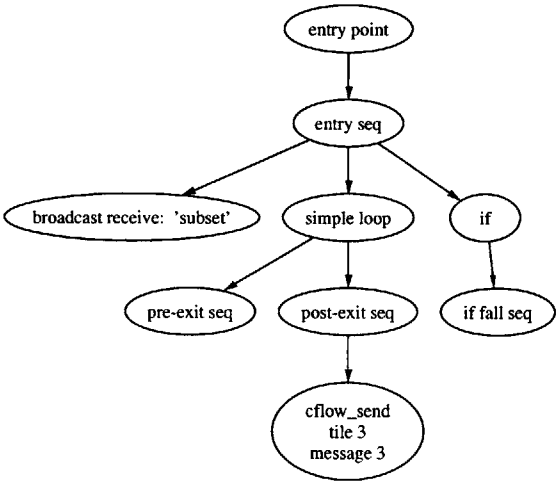
The unified message tree is a convenient method of merging the information contained separated message structure trees. Several properties of the UMT are particularly useful, including its deadlock-free ordering of communication transactions and the use of "minimal strand subsets" to reduce unnecessary control flow synchronization. The next section will provide a detailed description of an algorithm for constructing UMTs with these properties.

4.2 Constructing the Unified Message Tree

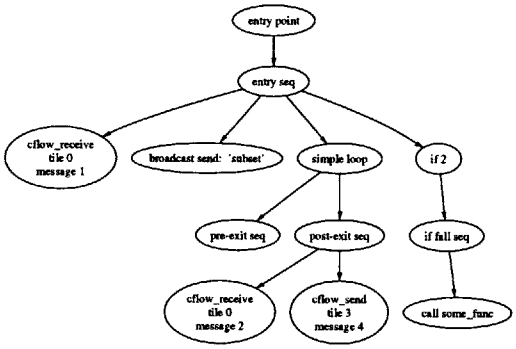
The construction of the unified message tree from the input message structure trees is done in three phases. The first phase, performed by the *buildLeaves()* function, matches the message send/receive pairs, broadcasts, and other leaf nodes in the MSTs. Each matched set of communication operations or function calls is assigned a single node in the new UMT.



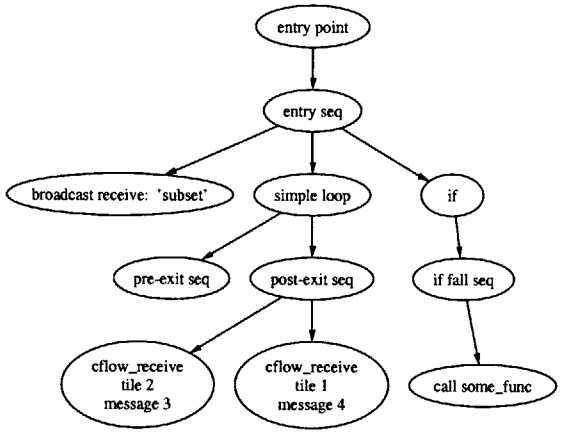
(a) Tile 0



(b) Tile 1



(c) Tile 2



(d) Tile 3

Figure 4-3: The message structure trees for each of the four strands in an application designed to make UMT construction difficult.

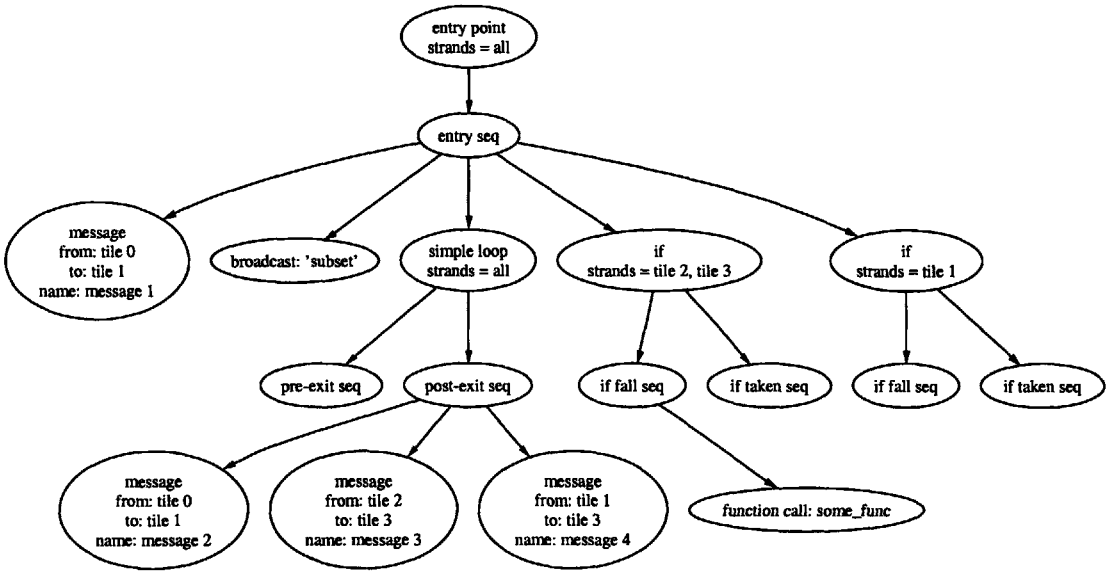


Figure 4-4: The merged message schedule for the difficult-to-merge application

The *buildLeaves()* function also guarantees that the program's communication patterns are deadlock-free.

The second phase in UMT construction, the *buildControlNodes()* function, creates all of the UMT's eventual branch nodes. It creates control region nodes with the correct "strand subset" and creates sequence nodes as children for those control region nodes. At the completion of the second phase, all the nodes in the final UMT have been constructed, but the tree is incomplete because none of the sequence nodes have had children assigned.

The final phase of UMT construction is performed by the *buildTree()* function. This phase attaches control region nodes and leaf nodes to their parent sequence nodes, finishing the unified message tree. Once the UMT is completed, the C-Flow compiler can use it to generate message routes, switch control flow, and eventually the output switch assembly code.

4.2.1 Data structures and Helper Functions

Before describing each of the three phase of UMT construction, we will first document some data structures and helper functions used by the main UMT algorithm. The following descriptions give an overview of each item, including a suggested implementation and runtime characteristics.

Traversing Message Structure Trees

Both the *buildLeaves()* and *buildTree()* functions perform a parallel traversal of all the input message structure trees. This traversal is performed in depth-first pre-order, so that a node is processed before its children and each individual child and its children are processed before the next child. For example, in the MST seen in Figure 4-3(a), the order of traversal is "entry point", "entry seq", "message 1", "simple loop", "pre-exit seq", "post-exit seq", "message 2".

Each *mst* object has a *mst.currentNode* property that indicates the current node in the traversal. The *mst.currentNode* property is initially set to the root node; calling the ***mst.start()*** function resets the current node to the root node. The ***mst.advance()*** function sets *mst.currentNode* to the next node in the depth-first pre-order traversal. When the traversal finishes, *mst.currentNode* is set to NULL. All of the above properties and functions can be accessed or executed in constant (O(1)) time.

Deadlock Avoidance Using Stacks

The *buildLeaves()* function uses a stack based method to guarantee that the input program's message schedule is deadlock free. This stack data structure has two operations: *void* ***push****(mst last, mst next)* and *mst* ***pop()***. The "last" argument to push specifies the MST to be pushed onto the stack, and "next" specifies the next MST that the *buildLeaves()* function will examine. Each *mst* object includes a *mst.onStack* property, initially false, that is set to true when "last" is pushed on the stack. If either "last" or "next" is already on the stack, a deadlock condition has been discovered and an error is reported. The reasoning behind this is explained in Section 4.2.2.

The pop() function simply returns the *mst* object on top of the stack and then decrements the top pointer. As will be explained in Section 4.2.2, the combined push() and pop() functionality can be used to store a list of MSTs whose traversal cannot currently progress in order to return to them later, after some dependancy has been resolved. Both the push() and pop() functions can execute in constant (O(1)) time.

Match Lists

The *buildLeaves()* function also determines the "minimal strand subset" for each control region in the UMT, so that *buildControlNodes()* can create UMT control region nodes with the correct *umt_node.strands* property (explained in Section 4.1.2 and shown in Figure 4-4). As indicated in Section 4.1.3, *buildLeaves()* initially marks each MST control region as being separate. As messaging commands descendent from that control region are discovered, the control region may have other "strands" added to it.

The *matchList* object facilitates this process. Each *matchList* object has two properties: *matchList.members* and *matchList.umt_node*. The *matchList.members* property stores a list of all the MST control region nodes that will be merged together to form a single UMT control region node. The *matchList.umt_node* property is initially NULL but is set by the *buildControlNodes()* function after the members list is finalized. A *matchList* object is created using the ***newMatchList****(mst_node node)* function. The "node" argument is the first MST node in the members list; a given MST node may only be put into one *matchList* object.

As the *buildLeaves()* function executes, it will find control region nodes in different MSTs that represent the same UMT control region node. The ***mergeMatchLists****(matchList A, matchList B)* function helps merge these previously separate regions into one. This function appends *B.members* to *A.members* and then sets *B.members* to *A.members*. Thus, with a constant (O(1)) time operation, both input *matchList* objects have the same member list, which contains the union of their previous members. Since each MST node is added to only one member list, and the *mergeMatchLists()* function conserves list members, we can guarantee that the merged member lists contain no repeated members.

Fields in a MST node

Several properties of a merge structure tree node are used during the construction of the UMT. The *mst_node.matchList* property is used in MST control region nodes to keep track of which other control region nodes will be matched with that node. MST nodes representing point-to-point messages have a *mst_node.far* property that indicates the MST at the other end of a message pair. Finally, MST nodes representing operations that may occur on more than two strands (broadcasts, barriers, and function calls) have a *mst_node.group* property.

The group object provides a list of all the MSTs involved in a particular group operation, as well as functions for accessing that list.

Other Helper Functions

The final two helper functions are used by *buildLeaves()* to verify that the various components of a messaging operation can be merged into one UMT node. The first function, *bool* ***matchOps****(mst_node a, mst_node b)*, determines whether two MST nodes can be part of the same operation. For example, *matchOps()* returns true if 'a' is a send from tile 0 to tile 1 and 'b' is a receive at tile 1 from tile 0. Similarly, *matchOps()* returns true if 'a' and 'b' are part of the same broadcast, barrier, or function call.

The second helper function, *bool* ***matchParents****(mst_node a, mst_node b)*, guarantees that the two input nodes have the same control flow dependencies in the UMT. Psuedo-code for this function appears in Figure 4-5. When a send-receive pair or the various strands in a group operation are merged into a single UMT node, that resultant node must have a clear set of control region parents. The *matchParents()* function is used to scan upwards in each input MST, matching control regions as it goes. Whenever two ancestor nodes are found to be "matchable" (both loops, both ifs, etc.), their matchLists are merged so that they will be represented by the same control region node in the final UMT.

4.2.2 Stage One: buildLeaves()

The *buildLeaves()* function performs three important tasks. First, it matches communication operations in different MSTs that correspond to the same transaction, pairing sends with receives and making sure that function calls are made by all of the function's member tiles. Secondly, it makes sure that the program has at least one possible deadlock-free message ordering. Finally, the function determines which MST control region nodes will be merged in the final UMT. That is, it determines the "strand subset" for each UMT region node, determining which MST loops, for instance, are guaranteed to execute together.

A pseudo-code representation of *buildLeaves()* appears in Figure 4-6. At a high level, this first step of UMT construction simply runs through each MST in depth-first pre-order, matching messaging operations to their pairs in other MSTs. Sequence nodes are ignored during this phase of MST construction, and whenever the traversal reaches a control region node, the node is given a new matchList containing only itself.

```
bool matchParents(mst_node a, mst_node b)
{
  //check for root node cases
  if(a and b are both root nodes)
    return true;
  else if(a or b is a root node)
    return false;

  //get the parents
  pa = a.parentNode;                                                    10
  pb = b.parentNode;

  //make sure parents are the same region type (loop, if, or) or
  // the same sequence type (if fall seq, loop pre-exit seq, etc.)
  if(pa.type != pb.type)
    return false;

  if(pa (and pb) are control region nodes){
    //guarantee that this and all parent control regions match
    if(a.matchList == b.MatchList)                                      20
      return true;
    else if(matchParents(pa, pb)){
      //combine the control regions into one
      mergeMatchLists(pa.matchList, pb.matchList);
      return true;
    }
  }

  return false;
}                                                                       30
```

Figure 4-5: A helper procedure for guaranteeing that two nodes have the same parents in the final UMT.

```
buildLeaves()
{
  //initialization
  entryMatchList = newMatchList(NULL);

  //loop through all strands, advancing as possible until
  // all strands have currentNode == NULL
  strand = msts.first();
  while(strand != NULL){
    node = strand.currentNode();                                    10

    if(node == NULL)
      strand = msts.next(strand);
    else {
      //sequences in MSTs will be implicitly matched by parent node
      if(node is a Sequence node)
        strand.advance();

      //control flow regions are assigned an initial matchList
      if(node is a ControlFlow node){                               20
        node.matchList = newMatchList(node);
        if(node is an "entry point")
          mergeMatchLists(node.matchList, entryMatchList);
        strand.advance();
      }

      //messages
      if(node is a peer-to-peer message){
        far_strand = node.far;
        far_node = far_strand.currentNode();                        30
        if(matchOps(node, far_node)) {
          if(matchParents(node, far_node)){
            node.umt_node = far_node.umt_node = umt.nodeFromMST(node);
            strand.advance();
            far_strand.advance();
            if(far_strand.onStack)
              strand = pop();
            }
          else
            error("matching message ends have different control flow");  40
        }
        else {
          push(strand, far_strand);
          strand = far_strand;
        }
      }

      //all other nodes are broadcasts, barriers, or function calls
      else
        synchGroup(strand, node);                                   50
    }
  }
}
```

Figure 4-6: The first of three UMT construction functions matches communication operations in the different MSTs while guaranteeing deadlock avoidance. It also determines which control flow nodes are to be grouped together.

The real work of *buildLeaves()* happens when an MST communication node is reached. Whenever a send operation is found, the algorithm tries to find its matching receive, and vice versa. A "matching" operation satisfies two tests. First, *matchOps()* guarantees that the messaging operations and endpoints match. For example, a send from tile 0 to tile 7 only matches a receive on tile 7 from tile 0. Secondly, the *matchParents()* function, defined above, guarantees that the resulting UMT message operation has a unique set of control region dependencies. *MatchParents()* scans upwards from the message ops in each MST, making sure that the path from the root to the message op has the same order of control regions and sequences. If this is the case, each control region's *matchLists* are merged, so that they lead to a single UMT control region node, and the message ops are allowed to "match".

In order to guarantee deadlock avoidance, the current node (*node*) in the current MST (*strand*) must "match" the current node (*far_node*) in the MST at the message's other endpoint (*far_strand*). If a matching operation is not the next operation in the "far" MST traversal, some operation before the "match" could have a message dependency leading to deadlock.

The "stack" helper functions described above help in guaranteeing that both message operations in a pair can execute without external dependencies. If the *far_strand* is not currently at a matching message operation, the current MST (*strand*) is pushed onto the stack and *far_strand* is processed on the next main loop iteration. The main loop iterates, potentially pairing off other messages until it finds a message operation that "matches" an operation from an MST that is already on the stack.

When an operation matches a "far" MST that is already on the stack, there are two possible consequences. If the "far" MST traversal is currently at a node that matches the current message operation, the match is successful. A message node is created for the UMT, both traversals are advanced, and the stack is popped to return to the MST that was waiting for the current one to advance.

If, on the other hand, the "far" MST is not on a matching operation, we have a deadlock situation. The current MST cannot progress unless is completes a message pair with "far", and "far" is on the stack because it is waiting for a message operation in the current MST. If the operations are not equal, both strands will wait for each other to complete some operation, and deadlock is unavoidable. In this case, the *push()* function described above

Node Type	matchList
entry point	tile 0, tile 1, tile 2, tile 3
simple loop	tile 0, tile 1, tile 2, tile 3
if	tile 2, tile 3
if	tile 1

Table 4.1: The matchLists after buildLeaves() completes. Each matchList contains the MST nodes that will be combined to form a single UMT node; this table lists the MST for each element in a matchList, showing how the matchList eventually becomes the "strand subset" at each UMT control region node

will throw an error when we push the current MST onto the stack. In particular, *push()* throws an error because the "next" argument will be "far", which is already on the stack.

The process for guaranteeing that group operations like broadcasts and function calls will not deadlock is similar, but more involved. The *synchGroup()* helper function, shown in Figure 4-7, handles this case. We will avoid a detailed description of this function, except to indicate that it checks for deadlock by finding all MSTs matching a particular group operation and pushing them onto the stack. Once all the MST traversals have reached a node that matches the group operation, a UMT node is created for the operation and all of the MSTs are popped off of the stack.

If *buildLeaves()* completes without discovering an unavoidable messaging deadlock, all of the operation (leaf) nodes for the final UMT have been created. Every point-to-point message, broadcast, barrier, and function call is now represented by a node. For example, after running *buildLeaves()* on the complicated example program first shown in Figures 4-3 and 4-4, the UMT contains the nodes seen in Figure 4-8. The final *matchList* objects are also determined at this point, leading to the lists shown in Table 4.1.

BuildLeaves() runs in O(N*h) time and O(N) space, where N is the total number of MST nodes and h is the maximum height of a MST. The function touches each MST node once, and at each node the *matchParents()* function can recurse all the way up from the current node to the root of the tree, touching each node in constant time. The Johnson paper [10] observes that larger programs tend to have broad and shallow trees, such that N grows much more quickly than h. If this is the case, the runtime is effectively O(N).

```
synchGroup(strand, node)
{
  group = node.group;
  if(group.bReached == false){
    group.bReached = true;
    group.firstStrand = strand;
    group.farStrand = group.start();
  }

  //first strand to arrive loops through all other strands in the
  // group operation, until all arrive at the operation
  if(node == group.firstMember){
    while(group.farStrand != NULL){
      far_node = group.farStrand.currentNode();
      if(matchOps(node, far_node)){
        if(matchParents(node, far_node)){
          nextStrand = group.next();
          push(farStrand, nextStrand);
          group.farStrand = nextStrand;
        }
        else {
          error("matching group ops have different control flow");
        }
      }
      else {
        push(strand, group.farStrand);
        strand = group.farStrand;
      }
    }
    if(group.nextStrand == NULL){
      //all strands have arrived at the group operation
      umt_node = umt.nodeFromMST(node);
      foreach memberStrand in group{
        memberStrand.umt_node = umt_node;
        memberStrand.advance();
      }
      while(top of stack is a strand in group){
        strand = pop();
      }
      //all group members are off stack, and strand is the strand
      // that first arrived at the group operation - we're done
    }
  }

  //if we're not the first strand to arrive, try to match first
  else {
    if(matchParents(node, group.firstStrand.currentNode())){
      strand = pop();
    }
    else {
      error("matched group ops have different control flow");
    }
  }
}
```

Figure 4-7: A helper function used by buildLeaves(). SynchGroup() guarantees deadlock avoidance for group operations like broadcasts and function calls.

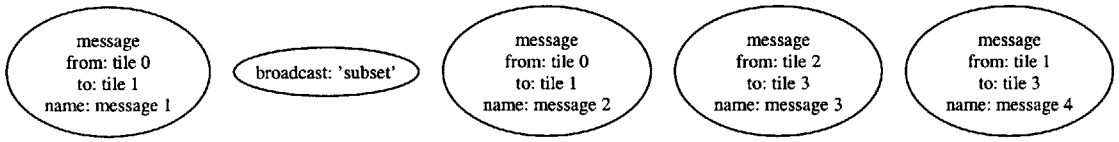


Figure 4-8: The progress of UMT construction after buildLeaves()

4.2.3 Stage Two: buildControlNodes()

The *buildControlNodes()* function, shown in Figure 4-9, creates the control region and sequence nodes for the final UMT. It accomplishes this task by scanning every node in every input MST and performing a set of tasks for each MST control region node it finds.

Each MST control region node must be matched with a UMT control region node. If the current MST node is the first in its *matchList* to be touched, no matching UMT region node exists, so one is created. As part of that creation process, a sequence node is created on the UMT node for each child sequence node of the MST node. Once the matching UMT node has been found, either by creating a new one or by looking on up in *mst_node.matchList.umt_node*, the MST node keeps a reference to it in *mst_node.umt_node*. Its child sequences also keep a reference to their corresponding UMT sequence nodes, using the same *mst_node.umt_node* property.

At the end of this procedure, all of the control region and sequence nodes in the UMT have been created. Also, every node in every MST has its *mst_node.umt_node* property set to its matching UMT node. The results of running *buildControlNodes()* on our example program appear in Figure 4-10; all that remains now is to assign children to the sequence nodes. The *buildControlNodes()* function clearly performs a constant time operation on each MST node, so it runs in O(N) time.

4.2.4 Stage Three: buildTree()

The final stage of UMT construction assigns children to each of the UMT sequences nodes, completing the tree. Determining the parent sequence for each message, broadcast, function call, or control region node is straightforward - the parent node in the MST has a reference to the appropriate UMT parent node stored in *mst_node.umt_node*. However, simply tracing the parentage of each non-sequence node is not sufficient; the child nodes must be added to their parent sequence nodes in a deadlock-free order.

```
buildControlNodes()
{
  foreach strand in msts{
    foreach node in strand{
      if(node is a ControlFlow node){
        //each matchList maps to a single node in the UMT
        if(node.matchList.umt_node == NULL){
          //the first time we touch a matchList, make a new UMT node
          umt_node = umt.nodeFromMST(node);
          node.matchList.umt_node = umt_node;                          10

          //the node keeps a record of which strands use it
          umt_node.strands = the MSTs of node.matchList.members;
          //make a child sequence for each one on the MST node
          foreach child of node{
            umt_seq = umt_node.newChildSeq(child.type);
          }
        }

        //every MST node keeps a reference to its matching UMT node   20
        node.umt_node = node.matchList.umt_node;
        foreach child of node{
          set child.umt_node to the matching child of node.umt_node;
        }
      }
    }
  }
}
```

Figure 4-9: BuildControlNodes() takes the final control flow region matchLists and builds all the UMT control flow region nodes. When this function completes, all the UMT nodes for messages and control flow have been built. The buildTree() function will assemble these nodes into the final UMT

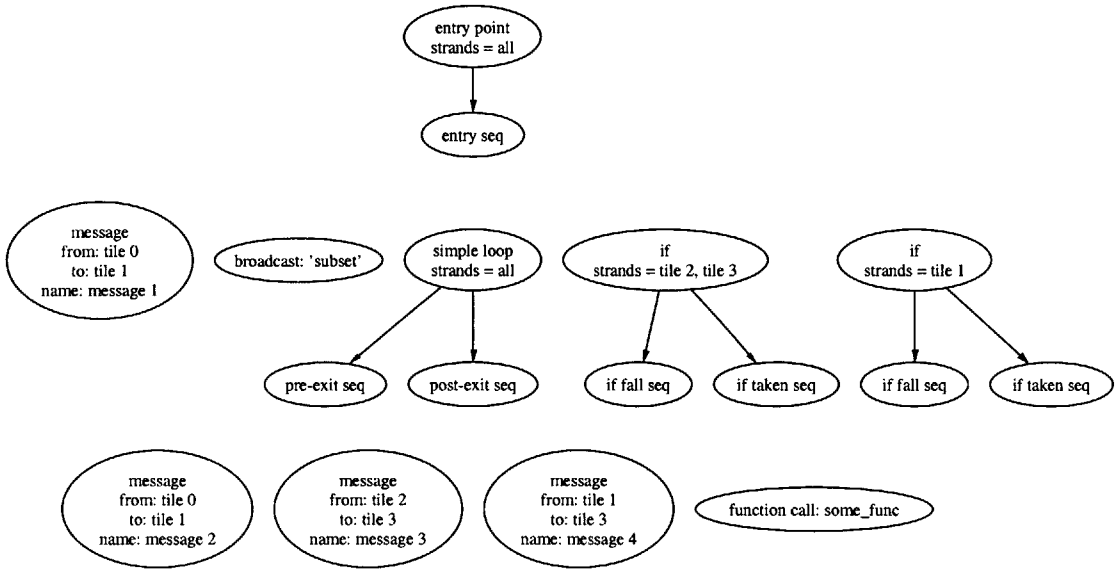


Figure 4-10: The progress of UMT construction after buildControlNodes()

The *buildTree()* function, shown in Figure 4-11, determines a deadlock free message ordering and attaches child nodes in that order. The deadlock free ordering is determined by traversing each MST in depth-first pre-order. As each node in the MST is reached, all the other MSTs that have a node "matched" with that node are advanced until they have reached their matching node. Thus, if a loop exists in three strands, all three MSTs are advanced until they have arrived at that loop. Similar advancement rules are applied to point-to-point messages and group communication functions, as well.

Once all of the strands matching a particular UMT node have advanced to the MST node that matches that UMT node, the operation represented by that UMT node can be executed without risking deadlock. Consider the two opposite endpoints of a point-to-point message. If the sending MST is ready to execute the send, and the receiving MST is ready to execute the receive, there are no external message dependencies that could lead to deadlock.

Assured that a particular UMT node can be executed without deadlock, the UMT node can be attached to its parent sequence. Since nodes are only attached to their parent sequences if they can execute without deadlock, the ordering of child nodes in every sequence must be deadlock-free. As long as the *buildTree()* function finishes, we can be assured that the generated UMT is deadlock-free. Thankfully, the *buildLeaves()* function has already verified that some deadlock-free schedule exists. Consequently, there is always at least one

deadlock-free UMT node that could be added at any given point during execution, and *buildTree()* is guaranteed to finish. The runtime of *buildTree()* is O(N), since every call to *traverse()* takes exactly one step along the traversal of some MST.

After the completion of *buildTree()*, UMT construction is complete. In the case of our example program, the final UMT appears in Figure 4-4. The overall runtime is O(N * h), dominated by the *buildLeaves()* function. If larger programs reach some maximum h, as suggested by [10], this becomes O(N). Regardless, the UMT construction process requires O(N) space. Our implementation of the UMT construction algirthm requires about 1,000 lines of C++ code. With the UMT complete, the C-Flow compiler can proceed to routing and switch code generation.

4.3 Routing and Switch Code Generation

The unified message tree provides a convenient intermediate form for message routing and switch code generation. The process of generating switch code requires two more, per-tile intermediate representations: the route structure tree and the switch control flow graph. This section describes each of these intermediate forms and details how they are used to finish switch code generation.

The route structure tree (RST) describes the code that should execute on a switch in much the way that a message structure tree describes the code that runs on the main processor. Like the MST, the RST's branch nodes represent control regions and sequences, describing the program's control flow hierarchy. However, the leaf nodes in an RST are individual route steps, not message operations. The routing operations include in a particular switch's RST represent both messages sent and received from that tile, as well as any messages that route through that switch.

The four route structure trees for the example UMT in Figure 4-4 appear in Figure 4-12. Notice that each tree contains only the control flow that actually executes on that tile's switch; for instance, tile 0 does no finish with an "if" region because the if regions in the UMT include only tiles 1, 2, and 3. Tile 0 also includes an example of a "through route". Its first routing operation, "$csto to $cEo", is part of "message 1" in the UMT, which goes from tile 0 to tile 1. The second routing operation, "$cEi to $cSo" is part of the "broadcast 'subset' " operation, which sends a word from tile 1 to tiles 2 and 3. Since the program

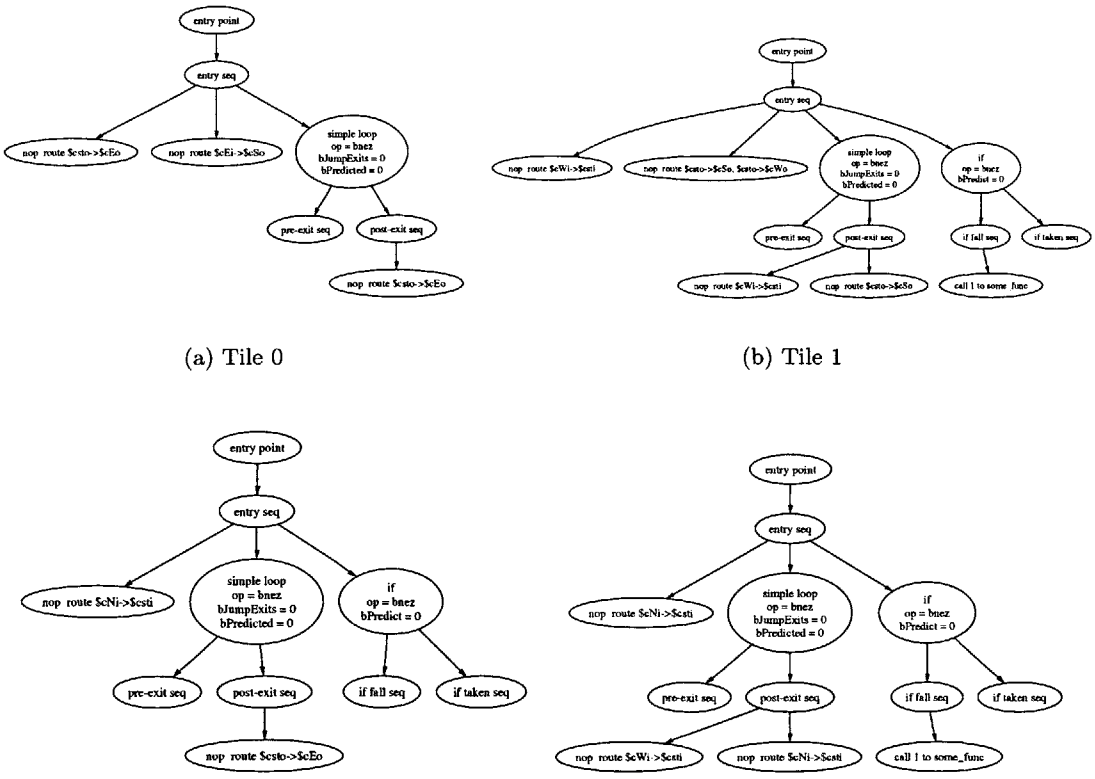
```
buildTree()
{
  //prepare a new depth-first, pre-order traversal
  foreach strand in msts {
    strand.start();
  }
  //push each strand until the traversal is complete
  foreach strand in msts {
    traverse(strand, NULL);
  }
}

traverse(mst strand, umt_node stop)
{
  //check if we've advanced as far as requested
  node = strand.currentNode;
  if((stop == NULL and node == NULL) or
     (stop == node.umt_node))
    return;

  //:otherwise, we need to continue traversing 'strand'
  if(node is a sequence node)
    strand.advance();
  else if(node is an MST control region) {
    //attach control regions to parent sequence nodes
    foreach other in node.strands {
      traverse(other, node.umt_node);
    }
    attachToSequence(node);
    foreach other in node.strands {
      other.advance();
    }
  }
  else if(node is point-to-point message) {
    traverse(node.far, node.umt_node);
    attachToSequence(node);
    node.far.advance();
    strand.advance();
  }
  else { //node is a broadcast, barrier, or function call
    foreach other in node.group {
      traverse(other, node.umt_node);
    }
    attachToSequence(node);
    foreach other in node.group {
      other.advance();
    }
  }
}

attachToSequence(mst_node node) {
  //the parent in the MST has reference to the parent in the UMT
  umt_seq = node.parent.umt_node;
  umt_seq.addChild( node.umt_node );
}
```


Figure 4-11: BuildTree() assigns child nodes to sequence nodes in a deadlock-free order.



(a) Tile 0

(b) Tile 1

(c) Tile 2

(d) Tile 3

Figure 4-12: The routed structure trees for each of the tiles in our "challenging" application. At this point in the compiler, all message commands have been replaced by individual routing steps.

runs on a 2x2 grid, the router chooses to route from tile 1 (x=1,y=0) to tile 2 (x=0, y=1) by going through tile 0 (x=0,y=0).

A pseudo-code algorithm for generating the route structure trees appears in Figure 4-13. The *buildRouteTrees()* function runs through every node in the UMT, copying it into the per-switch RSTs if the operation depends on that switch. Thus, control regions are copied into an RST if that strand is in their "strand subset". Function calls are copied into every strand that has a copy of that function. Finally, messaging operations like point-to-point messages, broadcasts, and barriers are expanded into a series of route steps. These route steps are copied into whatever strand RST they must run on; thus, a route from tile 0 to tile 2 via tile 1 will copy routing operations onto tiles 0, 1, and 2.

Our routing algorithm first attempts dimension ordered routing in X-Y order. If a route

```
buildRouteTrees()
{
  //make an RST for every strand
  foreach strand in strands {
    strand.rst = new rst();
  }

  //copy only necessary UMT nodes into RSTs
  foreach node in UMT, in depth-first pre-order {
    if(node is a sequence node) {                                    10
      continue;
    }
    else if(node is a control region) {
      //node.strands indicates which tiles have this control flow
      foreach strand in node.strands {
        strand.rst.copy_into_tree( node );
        foreach child_seq of node {
          strand.rst.copy_into_tree(child_seq);
        }
      }                                                              20
    }
    else if(node is a function call){
      //function calls operate on a group of strands
      foreach strand in node.group {
        strand.rst.copy_into_tree( node );
      }
    }
    else {
      //the nodes is a p2p message, broadcast, or barrier
      route_steps = route(node);                                     30
      foreach step in route_steps {
        step.strand.rst.copy_route_step( step );
      }
    }
  }
}
```

Figure 4-13: BuildRouteTrees() takes the unified message tree and generates a route structure tree for each strand. The route structure tree is much like a message structure tree, but the leaf nodes are routing steps instead of message operations.

cannot be found, Y-X order is attempted, and then a brute-force search is used. If none of these routing algorithms can find a route between a message sender and its recipients, the *buildRouteTrees()* function will abort with an error message. Once a route structure tree has been created for each switch, the RSTs are converted into switch control flow graphs. As with tile CFGs, the switch CFGs represent every operation that will execute on the switch in terms of basic blocks and edges between them. Generating the CFG from the route structure tree is a straightforward reversal of the process described in [10], and will not be described in detail. Instead, we simply present Figure 4-14, which shows the results of converting tile 3's RST, first shown in Figure 4-12(d), into a switch CFG.

Understanding the switch code in Figure 4-14 requires an understanding of C-Flow's calling convention, which is as follows. The caller tile code first calls into the callee, which then sends the new switch IP to the switch. The caller switch code does a "jump $csto" operation to move to the callee switch code. When the function returns, a similar process occurs. The tile code executes a return operation, then sends the switch return pointer to the switch. The callee switch code executes a "jump $csto" operation at the end of each function, allowing the switch to jump to its return point.

The "jr $csto" operation in "sw_begin_3" corresponds to a function call - it receives the entry pointer for the function being called and jumps to it. The "jr $csto" operation in "sw_begin_4" is part of the function return process; it receives the return address from the tile code and jumps to that location. The purpose of the rest of the basic blocks in Figure 4-14 should be apparent; the function simply executes a loop, then calls a function if some condition is met.

Once a switch CFG has been generated for each tile, the C-Flow compiler converts the CFGs into switch assembly code and writes them as output. The tile binaries are modified to send and receive switch instruction pointers at function call and return points, then written as output. With the switch code generated and the tile programs rewritten, the standard Raw toolchain can be used to compile all the strands together into a single Raw program, which can then be run in the simulator or on the hardware.

This chapter has described the process by which C-Flow converts individual tile message structure trees into generated switch code. The MSTs are merged into a single unified message tree, which provides a deadlock-free, global message schedule. This global message schedule is then routed, generating a route structure tree for each tile. The RSTs are then

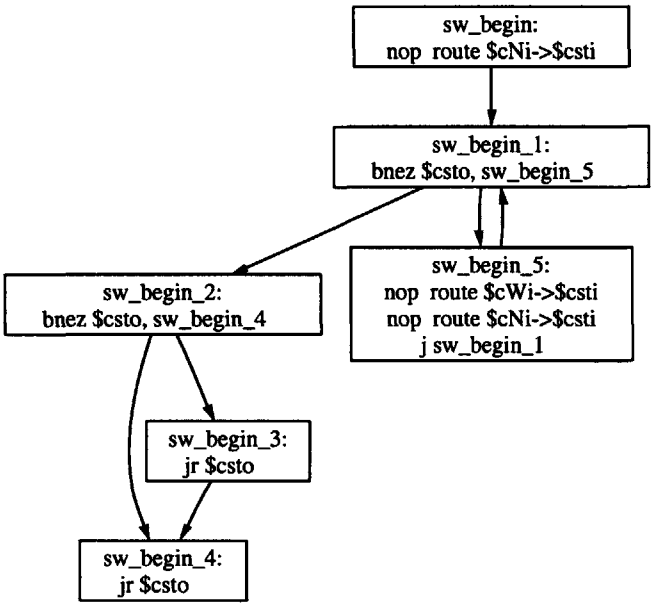


Figure 4-14: The final control flow graph generated for Tile 3.

converted into control flow graphs and finally switch assembly code, which is written into output files. The next chapter provides an overview of a few optimizations that may be applied to this process; a comparison of C-Flow's performance relative to hand-generated switch code appears in Chapter 6.

Chapter 5

Optimizations

The previous chapters have described the basic C-Flow compilation system. This baseline version is capable of merging separate program strands into a single, global, deadlock-free message schedule and then generating switch code for that schedule. The baseline compilation system provides a simple way to program Raw's static network, but it has several shortcomings in terms of performance. This chapter describes several optimizations that were added to the baseline C-Flow compiler in order to make its performance comparable to that of hand-optimized switch assembly code.

Our example program in this chapter is a four-tile version of the chroma-keying application first shown in Section 2.1.3. Designed for a two-by-two Raw grid, this implementation runs the same tile code on each processor and performs input and output on the right side of the chip. This geometry is pictured in Figure 5-1; the tile code is essentially the same as that first shown in Figure 2-2.

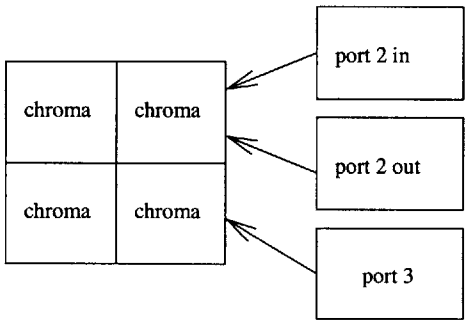


Figure 5-1: Chroma keying on a 2x2 tile configuration.

5.1 Dead Control Flow Elimination

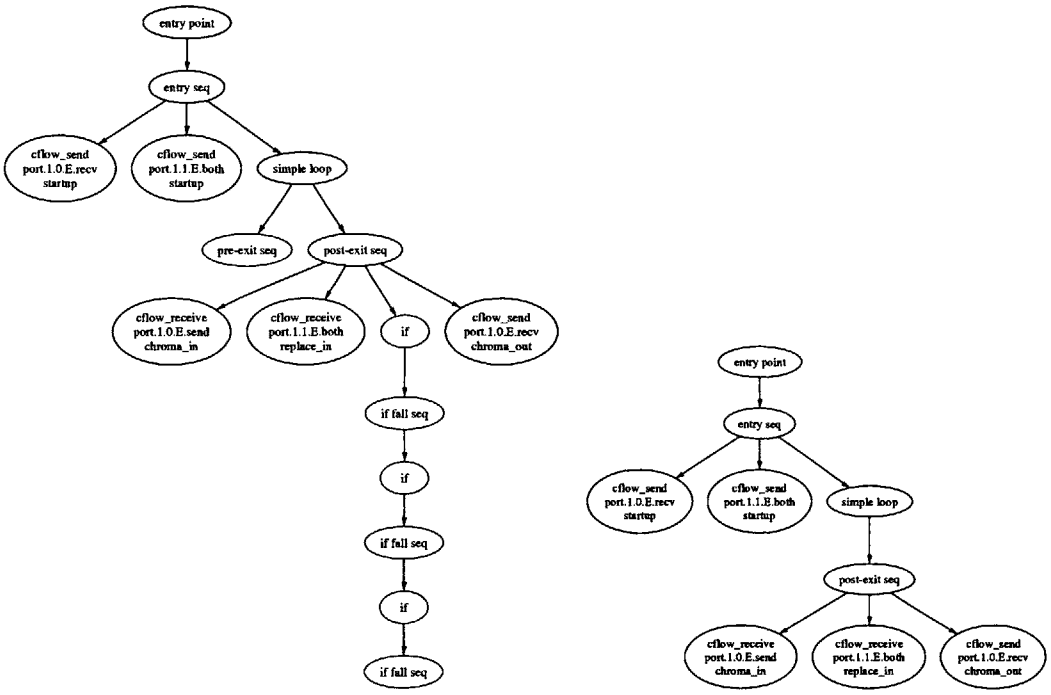
The baseline C-Flow compiler guarantees control flow synchronization by simply copying a tile processor's control flow onto its switch. This approach is easy to implement, but can result in unnecessary code being copied to the switch. If there are no message send, receive, or routing operations that depend on a particular control flow element, we refer to it as "dead control flow" and can eliminate it from the switch code without affecting program correctness.

Detecting dead control flow is straightforward, thanks to the message structure tree described in Section 3.6.3. By representing control flow regions and the operations that depend on them as a tree, the MST makes it easy to tell which control flow elements have dependent operations. For example, consider the MST in Figure 5-2(a). This MST represents the tile code for the example chroma-keying application. The body of the main loop has four elements: two pixel receive operations, an "if region" to select one of the pixels as a result, and a pixel send operation. Since none of the nodes in the "if region" are messaging operations, the switch can safely skip over that control flow region.

In the optimizing C-Flow compiler, dead control flow elimination is performed after a MST is generated for each strand and before the MSTs are merged into a unified message tree. The algorithm for removing dead switch control flow is based on a simple depth-first traversal of the MST. For each node in the traversal, a series of simple tests are performed to see whether it can be deleted from the tree. If the tests pass, the node is immediately deleted and the traversal continues at the next node.

The rules for deleting MST nodes are as follows. First, communication operations (send, receive, broadcast, barrier) may never be eliminated. Secondly, function call operations may be eliminated if the function being called does not affect the static network. In particular, if a function or any of the functions that it might call contain communication operations, the function call cannot be eliminated. Checking whether a function might call another function that uses the static network is done by a simple examination of the call graph.

The first two rules tell us which leaf nodes (communication operations and function calls) may be removed from the MST. The third and final rule allows us to propagate those deletions up the tree. In particular, a MST node representing a sequence or control flow region may be eliminated if it has no children. Since our depth first traversal guarantees that



(a) before elimination

(b) after elimination

Figure 5-2: The chroma keying tile program before and after dead switch control elimination

child nodes are checked before their parents, it is possible that a node that originally had children may have them all eliminated before it is examined, and thus nodes that originally had children may be deleted.

Performing dead control flow elimination on the chroma-keying tile strand results in the MST seen in Figure 5-2(b). The "if region" has been eliminated, which will eventually lead to three fewer branch operations in the switch code. However, the performance impact of this optimization is greater than simply the reduction in instruction count. Because the switch no longer needs to synchronize with the tile in order to receive branch conditions, messages from other tiles that route through the switch have greater scheduling flexibility. This reduction in unnecessary synchronization may actually have a larger performance impact than the simple reduction in instruction count.

5.2 An Improved Scheduling Algorithm

When C-Flow constructs a unified message tree from the input MSTs, it also assigns some deadlock-free ordering to each sequence node's child operations. Creating this deadlock-free ordering is valuable because it guarantees that the input program has matching send and receive operations for each message and that those operations can actually be run in some deadlock-free fashion. However, there may be more than one deadlock-free ordering of the operations in a particular sequence, and the ordering chosen by default may not be optimal.

Choosing an optimal ordering for a set of operations requires two types of information. First, a representation of the dependencies between operations is required in order to avoid deadlock. If a tile 1 performs a send followed by a receive, the receive is implicitly dependent on the send. If the switch code tried to perform the operations in the other order, it would not match up with the tile code and deadlock would result. The second type of information required for scheduling is some measurement of the time spent on each operation. By combining these two types of information, the scheduler can tell which operations can or should be executed before others.

C-Flow extracts timing information by counting the number of instructions between various operations. Data dependencies and operation latency are ignored; thus every instruction is assumed to take one cycle and to execute the cycle after the instruction that preceded it. This instruction counting information is added to the MSTs for each input

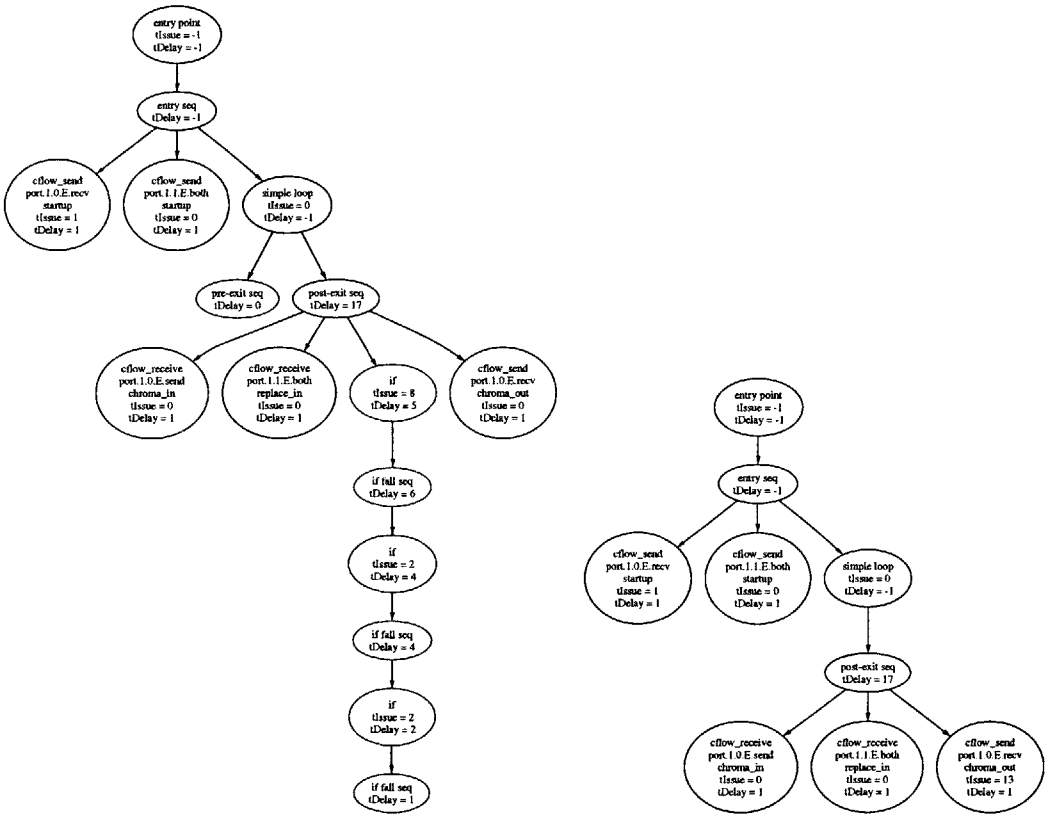
strand, as seen in Figure 5-3(a). Each node in the MST stores two fields: *tIssue* is the number of cycles between this operation and the completion of the previous operation, and *tDelay* is the number of cycles required to complete an operation. Control regions are assigned *tDelay* values based on their component sequences; for example, an "if region" takes two cycle plus the average of its two child sequence latencies. If *tDelay* cannot be determined, perhaps because the number of loop iterations is unknown, it is set to -1.

When dead switch control elimination is performed on the MST, the timing information must be preserved. This is accomplished by adding the *tDelay* and *tIssue* fields of any deleted nodes to the *tIssue* field of the next node. For instance, in Figure 5-3(b), a series of nested "if regions" has been eliminated. The sum of *tIssue* and *tDelay* for that region was 13 cycles, and that amount is added to the *tIssue* field of the send operation that followed the nested ifs.

The second component necessary for scheduling, dependency information, is stored as a "message dependence graph", or MDG. A graph is created for each sequence node in the UMT; thus, in our chroma-keying example, there is one graph for the sequence of operations that occurs in the body of the main loop. Nodes in the message dependence graph correspond to the UMT sequence's child nodes, so a node in the MDG can represent a control flow region, a messaging operation, or a function call.

Edges in the MDG represent "comes after" relationships. For example, a node representing a point-to-point message will have two incoming and two outgoing edges, connecting it to the operations that immediately proceed and follow it on each of the two strands involved in the message. Nodes representing control flow regions or function calls might have more edges, one incoming and one outgoing for each strand involved in the operation. Edges are also assigned a *tDelay* value, indicating the number of cycles between one operation and the next on a particular processor. This value is extracted from the timing information stored in the MST.

For example, consider the MDG for the main loop body of the two-by-two chroma-keying application, shown in Figure 5-4. The graph contains all of the information described above, as well as some extra scheduling-related fields that will be described in the coming paragraphs. The MDG has an entry and an exit node, representing the start and finish of the sequence in the UMT. Each of those two nodes has seven edges, one for each strand in the program (four tiles plus three I/O ports).



(a) before elimination

(b) after elimination

Figure 5-3: The chroma keying tile program with scheduling information. tIssue represents the delay between an operation and the operation which proceeded it; tDelay measures the number of cycles an operation takes to execute.

The other nodes in Figure 5-4 represent the messaging operations in the main loop. The bottom-most node is a point-to-point message between an input port and tile 0. It has two outgoing dependence edges, both going to the entry node. This indicates that the operation is the first one on both the input port and tile 0. The node also has two incoming dependencies, one from the next operation on tile 0 and the other from the next operation on the input port. In fact, there is a single path from exit to entry for each strand in the program, and each edge in the MDG represents a dependence in a single strand. Finally, we note that each edge has a "delay time"; in Figure 5-4 this value is the number on the right side of the additive pair associated with each edge. The left-hand value is the edge's "start time", which will be described next.

Once timing and dependency information have been extracted and stored in the MDG, the optimizing C-Flow compiler proceeds to its scheduling phase. The scheduler maintains a priority queue containing all operations whose dependencies have been satisfied. It selects the operation with the earliest "ready time" and routes it. Once the message has been scheduled, all of incoming dependence edges are marked as satisfied and given a "start time" - the cycle number on which the preceeding operation on that strand completed. Thus, when the scheduler routes a point-to-point message, one of the incoming edges is assigned a "start time" equal to the cycle after the message was sent, and the other is assigned the cycle after the message was received.

Once the dependency information and "start times" have been updated, the scheduler scan the graph for nodes whose dependencies are all satisifed. Any such nodes are assigned a "ready time" (*tReady* in Figure 5-4) and added to the scheduling priority queue. The value assigned to *tReady* depends on the types of operation; for messages it is the sum of the "start time" and "delay" on the edge representing the sending strand. For nodes representing control flow regions, *tReady* is the value of the largest sum on any outgoing dependence edge.

Thus, the scheduler in the optimizing compiler only schedules operation whose dependencies are satisfied. However, if more than one operation is ready, it selects the operation with the earliest "ready time" and schedules it. This can lead to a dramatically different scheduling order than the deadlock-free schedule initially assigned when the MSTs were merged into a single UMT. For example, Table 5.1 shows the message orderings chosen by the merging algorithm and the scheduler. The original schedule serializes the computation,

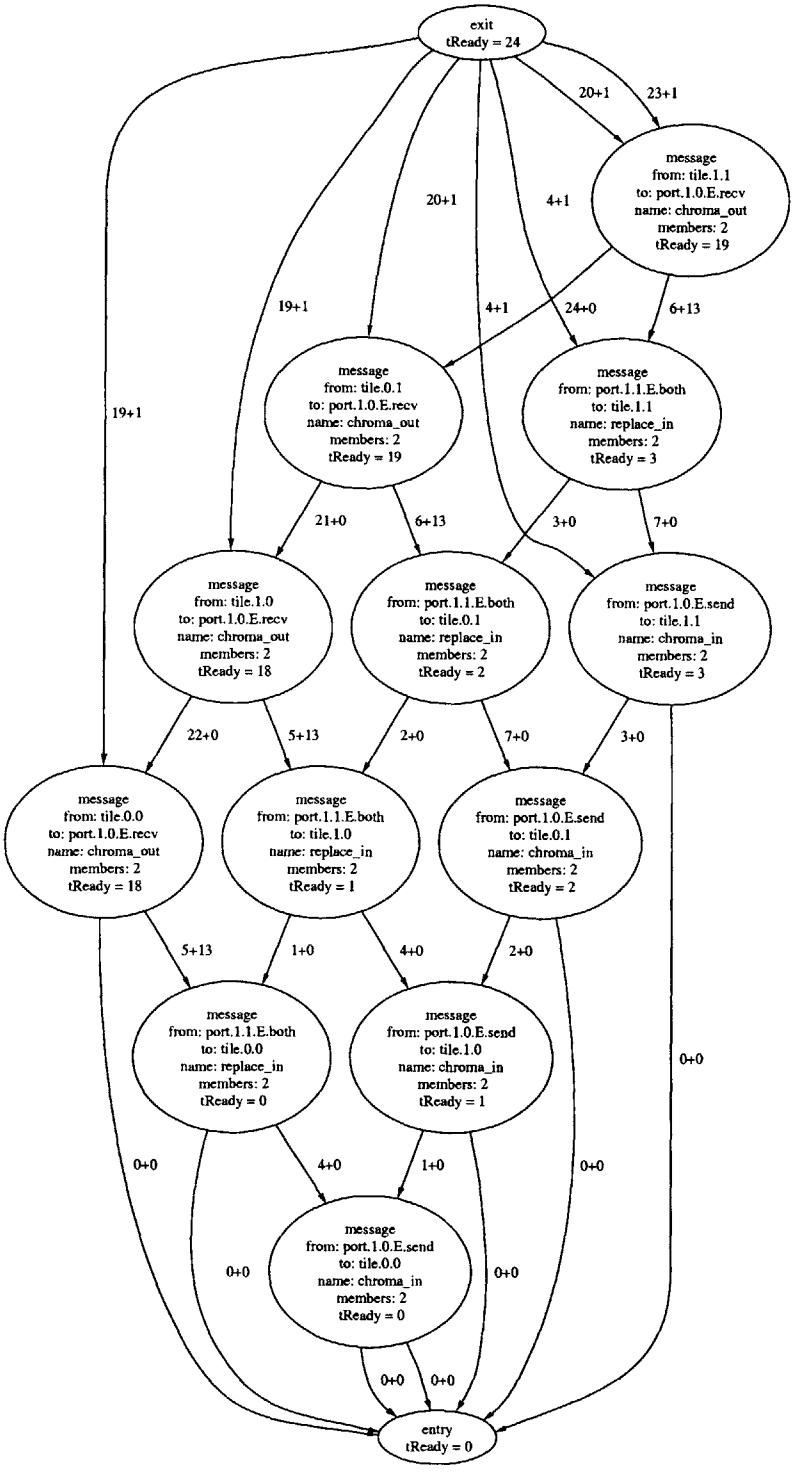


Figure 5-4: The message dependency graph for the chroma-keying application, along with associated timing and scheduling information.

Optimized schedule	Initial schedule
chroma_in: Tile 0	chroma_in: Tile 0
replace_in: Tile 0	replace_in: Tile 0
chroma_in: Tile 1	chroma_out: Tile 0
replace_in: Tile 1	chroma_in: Tile 1
chroma_in: Tile 2	replace_in: Tile 1
replace_in: Tile 2	chroma_out: Tile 1
chroma_in: Tile 3	chroma_in: Tile 2
replace_in: Tile 3	replace_in: Tile 2
chroma_out: Tile 0	chroma_out: Tile 2
chroma_out: Tile 1	chroma_in: Tile 3
chroma_out: Tile 2	replace_in: Tile 3
chroma_out: Tile 3	chroma_out: Tile 3

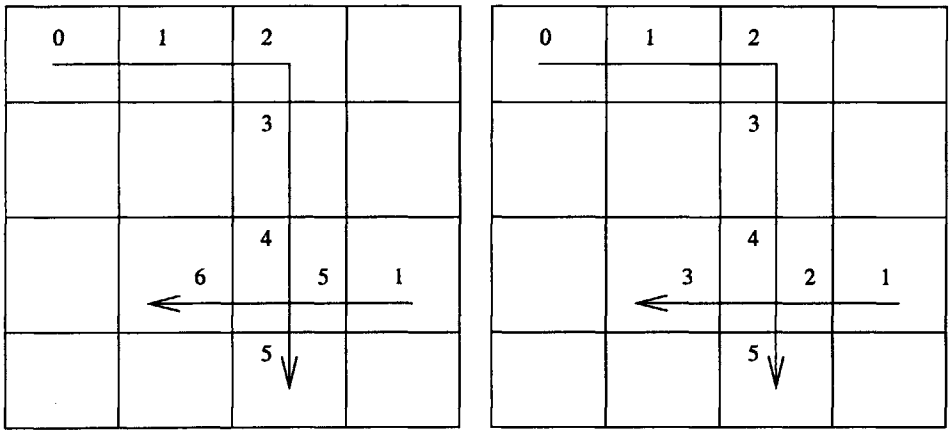
Table 5.1: The optimized and unoptimized message schedules for the chroma-keying main loop.

sending input and receiving results from each tile in turn. The optimized schedule does a much better job of finding parallelism by sending all of the inputs, allowing the computation to run, and then receiving all of the outputs. As will be seen in Chapter 6, the optimizing scheduler can lead to dramatic performance improvements.

5.3 Time Sensitive Routing

The improved C-Flow scheduler determines a "send time" for each message. This scheduling information is dependent on the latency between strand operations and the routing time of previous messages. However, a message does not necessarily transmit as soon as it is ready, because its routing instructions are inserted after any previously scheduled messages. Consider the example in Figure 5-5(a). This example shows two messages with overlapping routes. The first message, sent on cycle zero, has a very long route and doesn't reach the intersection point until cycle four. The second, sent on cycle one, has a shorter route and reaches the intersection at cycle two. However, because the first message is sent earlier, it is scheduled first and thus its route proceeds the second message at the intersection point. As a result, the second message must wait for the first, even though our scheduler information indicates that it arrives at the intersection before the first.

In order to fix such issues, the C-Flow router and scheduler were improved so that individual routing instructions were scheduled more carefully. In particular, the router orders instructions according to the time their data dependencies become available, instead



(a) unnecessary delay

(b) delay removed

Figure 5-5: Two overlapping messages routes. Time sensitive routing allows the short message to finish before the long, even though it is scheduled second.

of the time their message was sent. As a result, overlapping routes are scheduled as in Figure 5-5(b). In this case, the shorter message proceeds before the longer, even though it was sent later. As a result, the short message is received three cycles earlier, and the longer message is received at the same time. If the shorter message was on the critical path, the program's execution time should improve as a result of time sensitive routing.

Implementing time sensitive routing while avoiding deadlock conditions can be difficult for two reasons. First, our time sensitive router allows multiple data words to route through a switch on the same cycle, even though their message were scheduled separately. When doing so, care must be taken that the message do not require the same set of resources - if multiple routes attempt to use the same channel, they cannot be combined. Secondly, care must be taken to maintain the ordering of operations at route endpoints. The time sensitive router allows messages to arrive earlier than the default router, so it must take care to avoid moving a receive operation before any previous operations. Thankfully, the MDG graph provides the information necessary to avoid such message re-ordering, so the time sensitive router can succesfully avoid deadlock conditions.

5.4 Summary of Optimizations

This chapter has introduced three optimizations designed to improve the performance of C-Flow applications. Dead switch control elimination removes unnecessary control flow from the switch code, reducing the instruction count and leading to fewer synchronization points between switch and tile processors. An improved scheduler orders messaging operations so that messages that send earlier are scheduled first. Finally, time sensitive routing allows overlapping routes to proceed as quickly as possible by scheduling routing instructions according to data availability instead of send time. All of these optimizations have been implemented so as to improve performance while still guaranteeing deadlock avoidance.

Chapter 6

Performance Evaluation and Conclusion

This chapter examines the performance characteristics of C-Flow compiled applications. It compares C-Flow programs with their hand-coded equivalents, finding interesting results in both relative performance and scalability. Having characterized the performance of C-Flow generated switch code, the chapter concludes with some observations on both the final C-Flow system and potential future work.

6.1 Benchmarks

C-Flow allows programmers to choose how their application will be parallelized without having to write the communication code between processors. Because its programming interface is unique, requiring one input program per strand, finding an appropriate control for performance evaluation is somewhat difficult. We have chosen to compare performance against several hand-coded applications. This allows us to compare C-Flow with the best available application performance. Both C-Flow and hand-coded applications have been parallelized by the programmer, so the only differences should be in the performance of GCC's optimizer and the quality of the network code generated by C-Flow.

6.1.1 Comparison with Manual Implementations

Figure 6-1 compares C-Flow performance at various levels of optimization with a reference, hand-coded implementation. The three applications were chosen because their implemen-

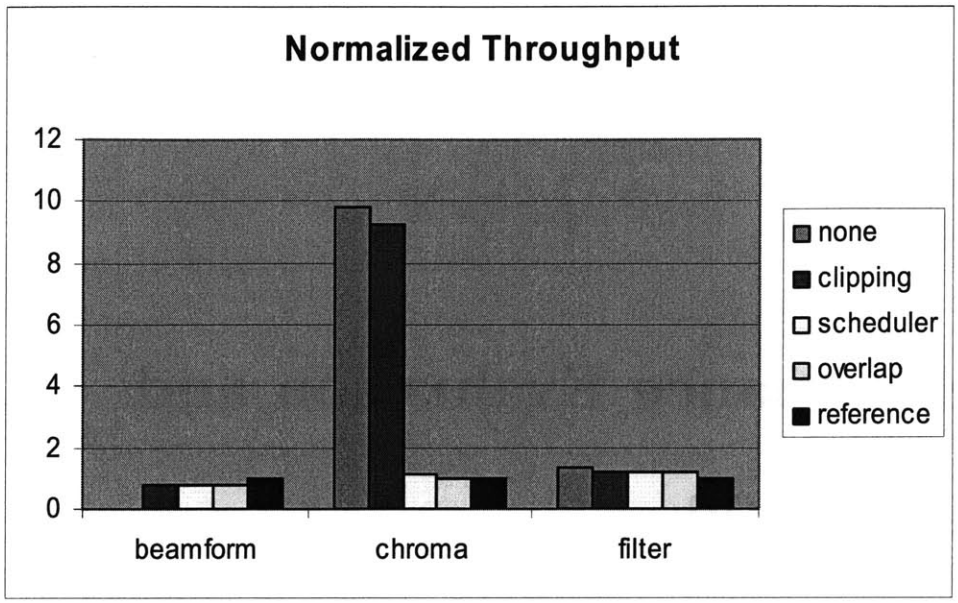


Figure 6-1:

tations are known to be well optimized and because the applications have differing network characteristics.

The beamforming application, implemented by Ken Steele and Eugene Weinstein, is part of the LOUD microphone array system [12]. The beamformer processes 50 megabytes of audio data per second, extracting a single voice from a crowded room. This application is characterized by long-distance communication, transmitting data from a single input port all the way across a four by four array. However, it also has heavy computational requirements, so the percentage of network bandwidth used is relatively low when compared to the other two benchmarks.

"Chroma" is a sixteen tile version of the chroma-keying application that has been used for demonstration purposes throughout this document. This application is characterized by long, overlapping routes from input ports to each tile and back. The computational requirements are relatively low, so the chroma-keying application is characterized by high network bandwidth usage and long, overlapping routes. These characteristics lead to a high degree of contention for network resources.

The "filter" benchmark performs a matrix convolution on an input image, accentuating

edges as in the first stage of an edge-detector. The implementation uses a "streaming" algorithm, routing data from a single input port to a single output port while performing a small amount of computation on each tile. This application is characterized by high network bandwidth usage but is dominated by short, mostly nearest neighbor routes. Thus, it requires an efficient method of sending or receiving data but does not involve much network contention.

Two results are immediately apparent from Figure 6-1. First, the C-Flow implementation of the beamformer is almost twenty-five percent faster than the hand-optimized version. This performance improvement is a result of the difficulty of writing static network code; the beamformer is a large application (thousands of lines of code) and requires a large amount of routing code. C-Flow's brute force approach to routing performs better because hand-optimizing static network code can take a lot of programmer time and effort.

Secondly, the optimizing scheduler allows a dramatic improvement in the performance of the chroma-keying benchmark. Without careful scheduling, the benchmark takes a ten-fold performance hit; with an optimized schedule its performance is within twenty percent of reference. In fact, if time sensitive routing is used in order to allow overlapping message routes, the C-Flow implementation is very slightly faster than the reference version.

The other optimizations, dead control flow elimination and time senstive routing, both show marginal performance benefits. Dead control flow elimination has clear benefits in both the chroma-keying and filter benchmarks, but the improvement is about ten percent. Time sensitive routing improves performance in the chroma-keying application because that benchmark is high in network contention. However, time sensitive routing also causes a very small degradation in "filter" performance, though using better scheduling metrics than simple instruction counting might eliminate the performance hit.

Thus, having compared results over benchmarks with different degrees of network contention and bandwidth usage, we can see that C-Flow provides performance equivalent to hand-coding while allowing the programmer to avoid writing difficult static network code. In fact, with all optimizations enabled, C-Flow actually yields better performance in two of our three benchmarks, and is off by about twenty percent in the third.

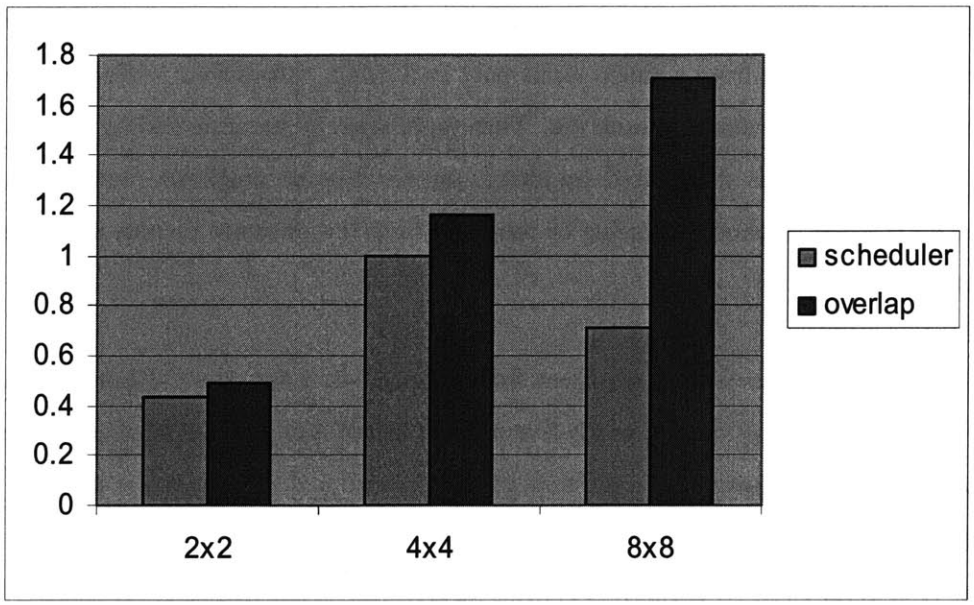


Figure 6-2:

6.1.2 Scalability and Optimization

While C-Flow delivers admirable results on sixteen tile configurations, an examination of performance on larger tile arrays is worthwhile, given that future processors will likely feature hundreds if not thousands of tiles. To this end, we have developed versions of the chroma-keying application that run on two-by-two, four-by-four, and eight-by-eight Raw grids. While this application is quite simplistic, the results give some indication of C-Flow's ability to scale to larger grid sizes.

Results for the four, sixteen, and sixty-four tile benchmarks appear in Figure 6-2. Two levels of optimization are shown; the first with the optimizing scheduler enabled, and the second with both the optimizing scheduler and time-sensitive routing. The application clearly shows better scalability when time-sensitive routing is enabled, particularly in the sixty-four tile case. In fact, the eight-by-eight grid actually performs worse than the four-by-four if time-sensitive routing is not enabled.

These results are clearly a consequence of network saturation at the I/O ports. The chroma-keying application only uses two network ports, and those ports become saturated

as the grid size increases. This network saturation has two consequences. First, performance does not scale linearly with the number of tiles because I/O bandwidth is constrained. Second, time-sensitive routing is critical for making the most effective usage of limited communication bandwidth. While the chroma-keying application is a somewhat contrived case (it would scale much better if multiple I/O ports were used), these results clearly indicate the time-sensitive and overlapped routing is critical to maintain high performance in applications with high network contention.

6.2 Future Work

Future work on C-Flow or other statically-scheduled message passing APIs could include more code optimization or better language integration. First, C-Flow currently uses temporary registers to store values going to and from the network, even though Raw has a register-mapped network interface. Some simple peephole optimizations could avoid these temporary register assignments, potentially bringing performance on the "filter" benchmark in line with that of the reference version. These peephole optimizations would be difficult in the current C-Flow architecture, though, because the binary rewriter infrastructure does not provide any sort of liveness information about the temporary registers [13].

More complicated optimizations could focus on using Raw's second static network. Using the second network would double the available communication bandwidth, leading to performance improvements in applications with high network usage or long, intersecting routes. Scheduling messages on the second network could be handled by the time-sensitive router, which already has the information necessary to detect network contention. However, this optimization would also prove difficult in the current C-Flow framework because it would require extensive rewriting of the tile code to connect with different network ports. Nonetheless, using the second network could lead to marked performance improvements, particularly in conjunction with peephole register optimizations.

Finally, research into software pipelining for distributed computation could also lead to dramatic performance improvements. Software pipelining techniques, such as modulo scheduling [14], could do an excellent job of pipelining network latency in order to improve parallel loop performance. C-Flow does not allow the programmer to specify software pipelining manually, since sends and receives must match within loop bodies so that writing

prologue and epilogue code would be impossible. However, the compiler should be able to perform software pipelining automatically, and it should have better information about network latency than does the programmer.

C-Flow's language interface could also be dramatically improved. The decision to base the compiler on Raw's binary rewriting infrastructure allowed quick implementation, but such an approach may prove inadequate for more serious use. Binary recompilation is appealing, particularly if it could allow multiple language interfaces to C-Flow, but the actual compiler implementation is so dependent on characteristics of GCC that such multi-lingual usage is probably infeasible.

Making C-Flow primitives a part of the language frontend, instead of passing them through the compiler as macros, would have several benefits. First, optimizations like function inlining, which currently break the merging phase of the compiler, would be feasible. Secondly, statically computed information like tile coordinates could be calculated as part of the source code, instead of having to create a separate source file for each tile. Finally, the peephole optimizations discussed above would come for free as part of the register allocation process.

Finally, the process of merging separate program strands into a single unified message tree could be made more flexible. At the moment, merging requires that the control flow nesting is identical in each input program. However, some reductions in the message structure tree could relax this requirement. For example, if statements in which both predicate sequences include the same network operation could be reduced to a single sequence, since the if doesn't actually affect network traffic. This would allow one strand to use an if statement to select a value to send to a second, while the second need only have a single receive statement to get either value.

6.3 Conclusion

The C-Flow compiler allows programmers to automatically generate static network code while maintaining performance equivalent to that obtained by hand-coding. It provides simple error detection, making sure that all of a program's send() and receive() operations match up with some other endpoint. C-Flow also guarantees that the generated network code will be deadlock free. These features should allow programmers to develop

static network applications much more quickly, while still maintaining excellent performance characteristics.

This thesis has introduced several key ideas. First, we have demonstrated that separate tile and port programs (strands) can be merged into a single, global message schedule in order to generate network code. While the merging algorithm will not work for arbitrary code, it can successfully merge any programs in which the input strands have matching control flow. We have also shown the importance of good scheduling metrics when generating static network code for applications with long, overlapping routes. Finally, time-sensitive routing has proven critical for applications which feature high degrees of network contention.

Bibliography

[1] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: On-chip interconnect for ilp in partitioned architectures. In *Proceedings of the INternational Symposium on High Performance Computer Architecture*, February 2003.

[2] M. Taylor, W. Lee, J. Miller, and D. Wentzlaff. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of International Symposium on Computer Architecture*, June 2004.

[3] Michael Taylor. The raw prototype design document, 2003.

[4] W. Lee, R. Barua, and M. Frank. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[5] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Water, and Wen mei W. Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual Int'l Symposium on Computer Architecture*, 1991.

[6] M. Gordon, W. Thies, and M. Karczmarek. A stream compiler for communication-exposed architectures. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[7] Henry Hoffmann, Volker Strumpen, and Anant Agarwal. Stream algorithms and architecture. Technical report, MIT, Laboratory for Computer Science, March 2003.

[8] MPI Forum. A message passing interface standard. Technical report, University of Tennessee, Knoxville, 1994.

[9] Anant Agarwal, Ricardo Bianchini, David Chaikent, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Limt, Kenneth Mackenzie, and Donald Yeung. The mit alewife machine: Architecture and performance. In *Proceeding of the Seventeenth Int'l Symposium on Computer Architecture*, 1991.

[10] et al R. Johnson, D. Pearson. The program structure tree: Computing control regions in linear time. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[11] K. Cooper, T. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice and Experience*, 2001.

[12] E. Weinstein, K. Steele, A. Agarwal, and J. Glass. Loud: A 1020-node modular microphone array and beamformer for intelligent computing spaces. Technical report, MIT, Laboratory for Computer Science, April 2004.

[13] Lawrence T. Kou. On live-dead analysis for global data-flow problems. *Journal of the ACM*, 1977.

[14] B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994.