The Admiral Language Proposal

Amir Kamil
Computer Science Division, University of California, Berkeley
kamil@cs.berkeley.edu

UCB-AK04

June 24, 2008

1 Introduction

The Fleet project is an attempt to design a new architecture around the constraints of modern chip fabrication methods, in which logic is cheap but communication is expensive. It consists of a set of concurrently executing functional units connected by a network. The programmer is given explicit control over communication between the functional units.

Since the Fleet architecture fundamentally differs from modern sequential and multicore machines, the languages used to program those machines are unsuitable for Fleet. Instead, a new language must be designed with the Fleet architecture in mind. This paper proposes the *Admiral* programming language. The goal of Admiral is to simplify Fleet programming while taking maximum advantage of the concurreny in a Fleet implementation.

2 Fleet Overview

Most modern computer architectures have evolved from the designs of many decades ago. At the time, chips consisted of relatively few transistors, so logic and storage were expensive while communication was not. The resulting designs were thus based around sequential streams of relatively complicated instructions, and performance was increased by making those instructions run faster through clock speed increases and other architectural improvements.

Modern processors, on the other hand, consist of many transistors, so logic and storage are now relatively cheap, but communication is relatively more expensive. Performance increases now come primarily from exploiting parallelism at both the instruction and the thread level. At their core, however, modern processors are still sequential designs, with parallelism tacked on afterwards.

Instead of trying to extend inherently sequential architectures with parallelism, it may be more productive to design a new architecture around the constraints of modern hardware implementation. The Fleet [2] project is an attempt at such an architecture that is inherently concurrent and in which control of communication is put in the hands of the programmer.

Figure 1 shows an example of a Fleet design. The main components are *ships*, the *switch fabric*, and *docks*.

2.1 Ships

The Fleet architecture includes a collection of functional units called *ships*, each of which may have any number of inputs and outputs. Example ships include adders, multipliers, multiplexers, FIFOs, and so on. The Fleet architecture

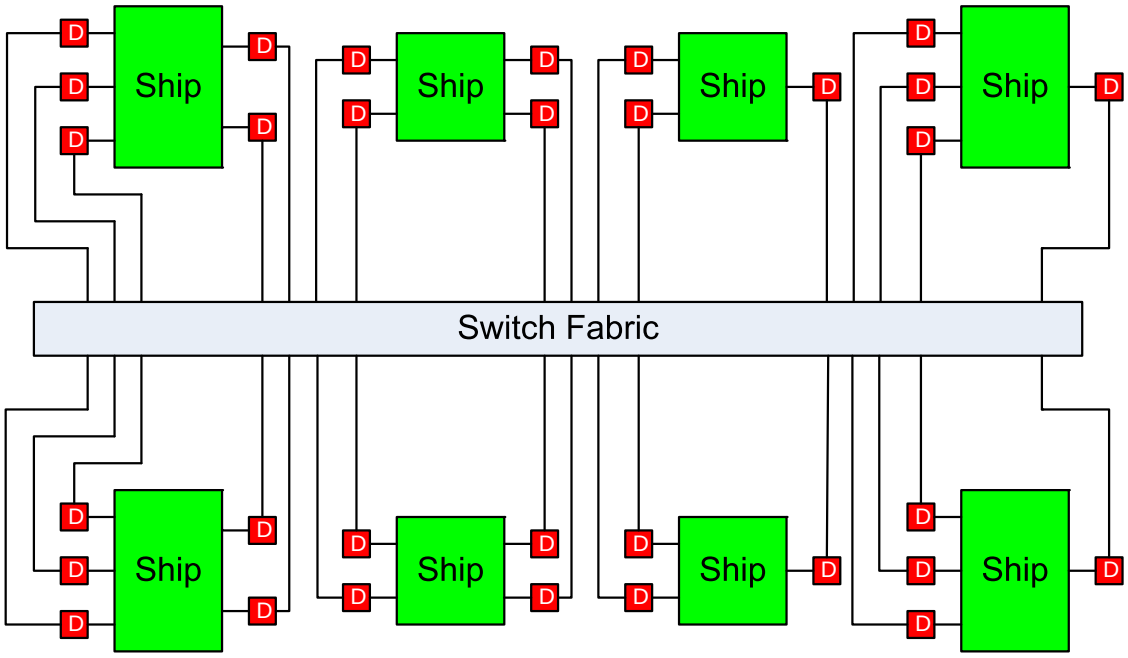

Figure 1: An example of a Fleet implementation.

itself is independent of the number of or types of functional untis, though an actual implementation must specify these details. This allows the architecture to scale through the addition of more functional units, or to be customized for a partcular application by changing which ships are included.

2.2 Switch Fabric

The Fleet architecture requires the existence of a network or *switch fabric* to alllow communication between ships. The design of the switch fabric is not specified in the architecture. The only constraint is that the switch fabric must deliver items sent between a single source and a single destination to the destination in the same order in which they leave the source.

The simplest switch fabric is a tree structure, as shown in Figure 2. Switch fabric stages may also contain storage for buffering data.

2.3 Docks

The Fleet architecture specifies a standardized interface between ships and the switch fabric, called a *dock*. Each input or output of a ship is connected to a corresponding input or output dock, which is in turn connected to the switch fabric, as shown in Figure 3.

The docks are the main control units in the Fleet architecture. They consist of a data latch, an instruction FIFO, and a number of connections to the switch fabric and ship. Input and output docks differ in their connections. An input dock has an incoming data connection from the switch fabric, an outgoing data connection to the ship, and an outgoing token connection to the switch fabric. An output dock has an incoming data connection from the ship, an outgoing

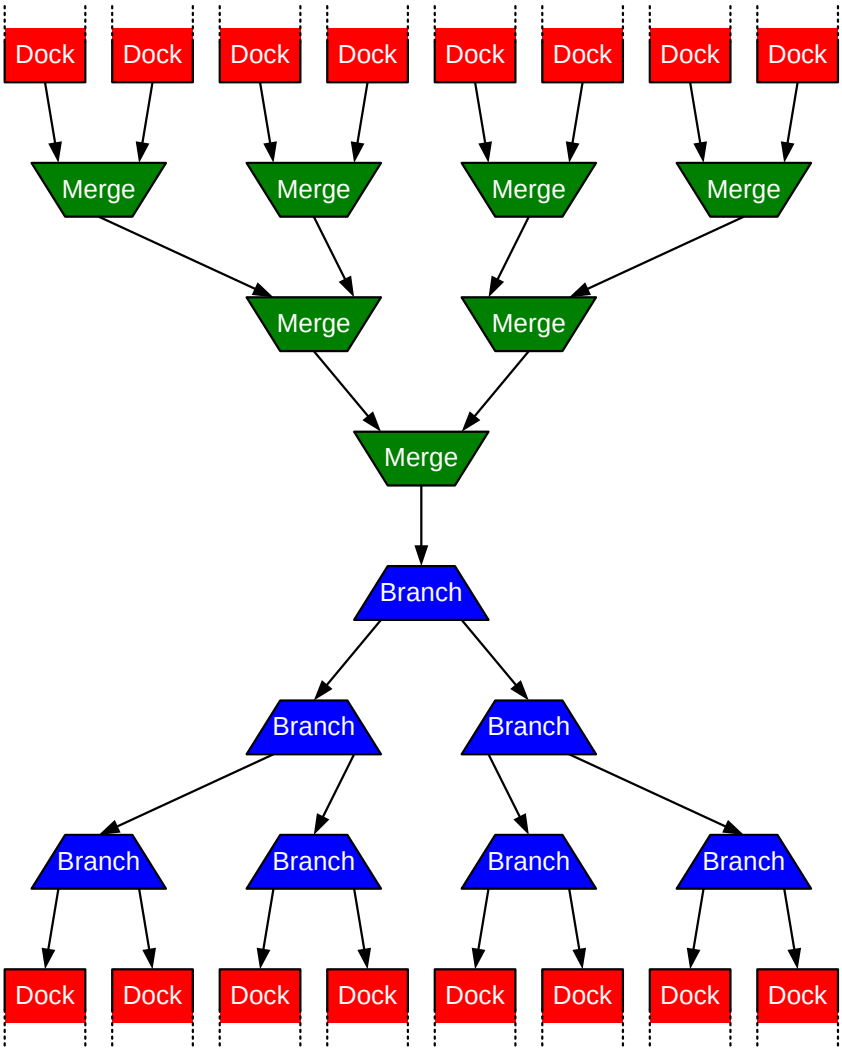

Figure 2: A simple switch fabric.

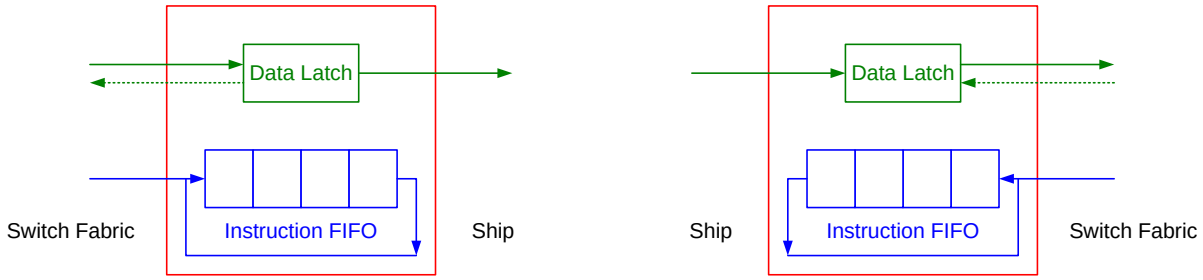

Figure 3: Input and output docks. Dashed lines correspond to token connections.

data connection to the switch fabric, and an incoming token connection from the switch fabric. Both dock types have an incoming instruction connection from the switch fabric.

2.4 Instructions

Data movement through a dock is controlled by a stream of instructions. Instructions arrive from the switch fabric, enter the instruction FIFO, and execute when they reach the end of the FIFO. Depending on the instruction type, it may execute once, a specified number of times, or forever (*standing* instructions) until it is externally destroyed. Alternatively, the instruction may execute once and then be requeued in the instruction FIFO.

Instructions in Fleet specifies whether to accept or discard data input to the dock, whether or not to latch the accepted data item, and whether or not to send the latched data item through the data output. In the case of an input dock, the data is sent to the ship, but for an output dock, the instruction specifies the destination dock. An instruction for an input dock can also send a token to a specified destination, while an output dock can wait for a token to arrive from the switch fabric before executing.

2.5 Fleet Programs

Fleet programs consist of a set of *codebags*, which are collections of instructions. For each instruction, the codebag specifies which dock it is to be executed at. A codebag is dispatched by sending its descriptor to a special *fetch* ship, which then injects all the instructions from the codebag into the switch fabric. The switch fabric then delivers the instruction to their respective docks. The only sequencing guarantee between instructions in a codebag is that instructions at a single dock arrive at that dock's instruction FIFO in the order in which they appear in the codebag.

A Fleet program contains a special codebag that is dispatched at program startup. Subsequent codebags are only dispatched when a previously dispatched codebag causes the new codebag's descriptor to arrive at the fetch ship.

3 Problems in Programing Fleet

While the architectural design of Fleet is quite simple, programming Fleet has proven to be quite difficult. This section discusses some of the reasons that this is the case.

3.1 Dock Operations

Docks provide a number of instructions involving their data latches and connections to their ships and the switch fabric. However, these instructions are all local to an individual dock, so a conceptually simple operation such as "move data from *A* to *B*" requires at least one instruction at each of docks *A* and *B*. A slightly more complicated operation such as "move data from *A* to both *B* and *C*" is even more difficult, requiring careful usage of the instruction FIFO at *A* to set up a loop alternating between *B* and *C*.

3.2 Sequencing

The only sequencing guarantees that Fleet provides are that data from a single source to a single destination will arrive in order, and that instructions for a single dock from a single codebag will arrive at the dock in the order in which they appear in the codebag. For many programs, these guarantees are insufficient. Tokens can be used to enforce sequencing when this is the case. For example, if dock *C* needs to receive alternating inputs from docks *A* and *B*, *A* and *B* can be programmed to wait for a token before sending data to *C*, and *C* can send a token to *A* or *B* when it receives data from the other. Depending on the sequencing pattern required, it may be difficult for the programmer to correctly set up the required token management.

3.3 Deadlock

Given the concurrent nature of Fleet, there are many ways to deadlock the system. Deadlock can arise from semantic errors, such as a circular dependency between a set of instructions. More insidious, however, are the kinds of deadlock that can occur due to implementation details of a particular Fleet design.

A Fleet implementation can only have a limited amount of queue space in the switch fabric, so one way to deadlock the system is to exhaust the queue space in a critical path through the switch fabric. For data traffic, a programmer must set up flow control using tokens in order to prevent this from happening. For example, if dock *A* is sending a lot of data to dock *B*, then *A* can be programmed to only send data when it receives a token, and *B* can be programmed to send a token to *A* each time it receives a data item. Then, the programmer can seed this connection with a number of tokens corresponding to how many items can be in flight at the same time. Increasing the number increases the amount of concurrency in the system but also increases the possibility of deadlock. Determining the correct number may be very difficult, and impossible to do without intimate knowledge of the rest of the program.

Flow control must also be done for instruction traffic, either by strictly limiting the number of instructions for a particular dock in each codebag or by queuing instructions in a data FIFO before dispatching them. Either option is tedious and can adversely affect the performance of a program.

3.4 Arbitrary Limits

A Fleet implementation also specifies limits that do affect program semantics. An instruction can either repeat or requeue forever or a small number of times, where the limit on this small number is defined by the Fleet implementation. A programmer must know what this limit is and work around it. The implementation also specifies the size of the instruction FIFO in a dock, which corresponds to the maximum size of an instruction loop. Again, the programmer must work around this limit.

3.5 Setup and Teardown

Setup and teardown for a repeating task can be quite difficult. In the flow control example in §3.3, docks *A* and *B* would normally be programmed with an instruction that repeats forever, known as a *standing* instruction. To stop the flow of

data from *A* to *B*, not only do the standing instructions need to be destroyed, but so do the tokens controlling the flow of data between the docks. This can be difficult to accomplish cleanly. Setup and teardown are also quite difficult for instruction loops.

3.6 Ship Management

Managing the collection of ships in a Fleet system poses some difficulties. A programmer must be careful not to use the same ship for different purposes simultaneously, or interference can arise. This may not be trivial, as it can be difficult to keep track of which ships are in use at each point in time. Even when a ship is no longer in use, if it contains internal state, the state must be cleared before the ship can be reused. Similarly, the state in each dock must also be reset before it can used again.

3.7 Topological Details

For a program to perform well, it may be necessary to take the topological of a Fleet implementation into account. For example, if a program requires ships of type S_1 and S_2 to communicate frequently and multiple ships of each type exist, then the choice of which S_1 and S_2 to use can affect performance. In particular, it may be beneficial to choose two such ships that are located in close proximity, assuming a switch fabric that takes advantage of this proximity. In other cases, it may be necessary to take advantage of *bypass paths* to reduce latency [1].

4 The Admiral Language

The Admiral language is an attempt to address the problems discussed in §3. It abstracts away some of the complex details of the Fleet architecture while remaining focused on data movement. Admiral is to Fleet as C is to a sequential machine.

Figure 4 shows the complete Admiral syntax. The different components of the language are discussed in the following sections.

4.1 Moves

The move statement is the most basic operation in Admiral. It specifies a source dock and a list of destination docks. A move operation causes a copy of the data item at the source to be sent to each of the destinations, while the item is also removed from the source.

There are two types of moves, *simple* and *counting* moves. Their syntax is as follows:

$$source -> destination\_list \quad \text{(simple)}$$
$$source -[n]-> destination\_list \quad \text{(counting)}$$

A simple move executes only a single time, while a counting move executes the specified number of times.

4.2 Blocks

Move statements are grouped into blocks. There are two types of blocks, *parallel* and *sequential*. In a parallel block, statements may execute in parallel, while in a sequential block, statements execute one after the other. Blocks may be nested, so it is legal to have a sequential block nested inside a parallel block, as in Figure 5. In this case, the statements in the sequential block will execute in sequence but in parallel to the rest of the statements in the parallel block.

$$
\begin{array}{rcl}
program & := & member^{*} \\
member & := & name\ \texttt{':'}\ block \\
 & | & shipdef \\
block & := & parblock \\
 & | & seqblock \\
parblock & := & \texttt{'\{'}\ vardecl^{*}\ (stmt\ \texttt{'||'})^{*}\ stmt\ \texttt{'||'}_{opt}\ \texttt{'\}'} \\
seqblock & := & \texttt{'\{'}\ vardecl^{*}\ (stmt\ \texttt{';;'})^{*}\ stmt\ \texttt{';;'}_{opt}\ \texttt{'\}'} \\
vardecl & := & shipname\ actuals_{opt}\ varname\ \texttt{';'} \\
actuals & := & \texttt{'<'}\ docks_{opt}\ \texttt{'->'}\ docks_{opt}\ \texttt{'>'} \\
stmt & := & block \\
 & | & move \\
move & := & source\ \texttt{'->'}\ docks \\
 & | & source\ \texttt{'-['}\ int\ \texttt{']->'}\ docks \\
source & := & int \\
 & | & name \\
 & | & constant \\
 & | & dock \\
docks & := & dock\ (\texttt{','}\ dock)^{*} \\
dock & := & name \\
 & | & name\ \texttt{'.'}\ name \\
constant & := & name\ \texttt{'.'}\ name \\
 & | & name\ \texttt{'.'}\ name\ \texttt{'.'}\ name \\
shipdef & := & \texttt{'ship'}\ name\ exports_{opt}\ \texttt{'('}\ imports_{opt}\ \texttt{')'}\ \texttt{'\{'}body\ \texttt{'\}'} \\
exports & := & \texttt{'<'}\ names_{opt}\ \texttt{'->'}\ names_{opt}\ \texttt{'>'} \\
imports & := & names_{opt}\ \texttt{'->'}\ names_{opt} \\
names & := & name\ (\texttt{','}\ name)^{*} \\
body & := & decls\ initblock_{opt}\ repblock_{opt}\ endblock_{opt} \\
decls & := & vardecl^{*}\ assign^{*} \\
assign & := & dock\ \texttt{'='}\ dock \\
initblock & := & \texttt{'init'}\ block \\
repblock & := & \texttt{'rep'}\ block \\
endblock & := & \texttt{'end'}\ block
\end{array}
$$

Figure 4: The Admiral sytax.

```
{
  a.b -> c.d ||
  {
    e.f -> g.h ;;
    i.j -> k.l ;;
  } ||
  m.n -> o.p
}
```

Figure 5: A sequential block nested inside a parallel block.

The type of a block is specified by the separator between move statements. For a parallel block, the separator is a pair of vertical bars, and for a sequential block, it is a pair of semicolons. The separator is optional after the last statement in a block.

4.3 Variable Declarations

Admiral programs must declare variable before using them. A variable declaration consists of a ship type, which can be virtual (§4.6), and a variable name, followed by a semicolon:

$$ship\_type\ var\_name\ ;$$

Variable declarations must occur at the beginning of a block, and the scope of a declaration is its enclosing block, including nested sub-blocks.

Ships are automatically managed by the Admiral compiler. They are allocated at variable declarations, and when a variable is no longer in scope, the associated ships are deallocated, and any internal state in either the ships or the docks is cleared. It is unclear at this point how to handle a situation where a program attempts to allocate a ship when all ships of that type are already in use.

4.4 Named Blocks

Admiral programs may contain named blocks at the top-level, using the following syntax:

$$name : block$$

Named blocks may be dispatched by sending their names to a fetch unit. The block named *main* is automatically dispatched at program startup.

4.5 Sources and Destinations

The source of a move statement can be an integer literal, a block name, a constant, or a dock. A ship-specific constant is specified by

$$ship\_type.constant$$

and a dock-specifc constant is specified by

$$ship\_type.dock\_name.constant$$

```
7:
  sendto mul.in2;
mul.in2:
  take;
  [*] deliver;
mul.in1:
  [*] take, deliver;
mul.out:
  [*] take, sendto add.in1;
13:
  sendto add.in2;
add.in2:
  take;
  [*] deliver;
add.in1;
  [*] take, deliver;
```

Figure 6: Fleet assembly for computing $7x + 13$. The programmer must send input to `mul.in1` and retrieve output from `add.out`, so this code is in passive mode. The code omits flow control and teardown of the standing instructions.

A dock is specified by

$$var\_name.dock\_name$$

Only docks may be the destinations of a move statement.

4.6 Virtual Ships

A common paradigm in programming Fleet is to connect multiple ships together using standing instructions in order to accomplish a higher-level operation. For example, if the polynomial $7x + 13$ needs to be computed for many values of x, this can be accomplished using a multiplier ship and an adder ship with standing instructions at the output of the multiplier and an input of the adder to constantly send data between them, using flow control to prevent flooding of the switch fabric. In addition, standing instructions can be placed at one of the inputs of the multiplier and the other input of the adder to constantly feed them the numbers 7 and 13, respectively. With these standing instructions in place, the program can consider the combination of the multiplier and adder to be a *virtual ship* with a single input and a single output and use it accordingly. Figure 6 shows an example of this, with the flow control omitted.

There are two modes in which programmers generally use the docks in a virtual ship. The first is the *active mode*, where data is continually transferred from a set of outside docks to the virtual inputs or from the virtual outputs to a set of outside docks. The second is the *passive mode*, in which the programmer manually sends data to the virtual inputs and retrieves data from the virtual outputs.

Admiral provides a facility for defining virtual ships that can be in active or passive mode, or a combination of the two. Figure 7 contains an example virtual ship, with the syntax is as follows:

$$\texttt{ship}\ ship\_name\ imports\ (\ exports\ )\ \{\ declarations\ init\_block\ repeat\_block\ end\_block\ \}$$

Here, *imports* is an optional set of names enclosed by angular brackets, and *exports* is also an optional set of names. The imports correspond to a ship in active mode, while the exports correspond to a ship in passive mode. Both imports and exports are specified as two comma-separated lists of user-provided names, with the lists separated by the move symbol $->$. The names to the left of this symbol correspond to inputs, while those to the right correspond to outputs.

```
ship Foo<a -> b>(in -> out) {
  Adder add1;
  Adder add2;
  in = add1.in1;
  out = add2.out;
  init {
    1 -> b ;;
    2 -> b
  }
  repeat {
    {
      a -> add1.in2 ||
      add1.out -> b
    } ;;
    a -> add1.in1, add1.in2
  }
  end {
    a -> b
  }
}
```

Figure 7: A virtual ship definition.

The *declarations* consist of ship variable declarations and assignments indicating which real docks the exported virtual docks correspond to. All component ships must be declared here, and variable declarations are illegal in the rest of the virtual ship definition. In addition, all exported docks must be assigned here. The syntax for an assignment is as follows:

$$dock1 \;=\; dock2\,;$$

One side of the assignment must be an imported dock or a dock of an internally declared ship, and the other side must be an exported dock.

The *init_block* is a block of code to be run when the virtual ship is allocated.

The *repeat_block* is the steady-state behavior of the virtual ship. It executes repeatedly until the virtual ship is deallocated. Only one iteration of this block can be active at any time. At the moment, it is not clear whether or not an iteration should begin immediately after the previous one or until all inputs are available. The former is more efficient but can result in an incomplete final iteration.

The *end_block* executes when the virtual ship is deallocated.

A virtual ship to compute $7x + 13$ is shown in Figure 8. No flow control or teardown code needs to be provided, since the Admiral compiler will generate it automatically.

A virtual ship variable can be declared like a real ship, except that if the virtual ship specifies imports, then the imports must be specified as existing docks at the time of declaration. The virtual ship can then actively retrieve data from and send data to these docks. In addition, the programmer can send data to or retrieve data from the exports of a virtual ship as if they were real docks, thus using the virtual ship in passive mode. Figure 9 shows an example of declaring and using virtual ships.

In general, it appears that standing instructions are primarily used by programmers to create virtual ships. As discussed in §3, setting up and tearing down standing instructions can be quite difficult. Thus, Admiral provides virtual

```
ship Poly(in -> out) {
  Multiplier mul;
  Adder add;
  in = mul.in1;
  out = add.out;
  repeat {
    7 -> mul.in2 ||
    mul.out -> add.in1 ||
    13 -> add.in2
  }
}
```

Figure 8: A passive virtual ship that computes $7x + 13$.

```
Adder a; // real ship
Foo<a.in1 -> a.out> f; // virtual ship with imports
Poly p; // virtual ship without imports
13 -> p.in;
p.out -> f.in;
```

Figure 9: Declaration and use of virtual ship variables.

ships instead of standing instructions, so that setup can be done automatically by the compiler when a virtual ship is declared and teardown when it is no longer in use. Admiral, however, does not reset ships with internal state, so the code to do so must be provided by the programmer in the *end_block*.

5 Compilation Strategy

Admiral relieves programmers of the difficulties discussed in §3 by moving the responsibility to deal with them to the Admiral compiler.

5.1 Move Statements

Instead of exposing dock operations, Admiral only has move statements, which the compiler then translates into the appropriate dock operations at the source and destinations, as demonstrated in Figure 10. Simple moves will be translated into one or more dock instructions at each end, while counting moves will be translated into either repeating or requeuing instructions at each end. Counts that are higher than provided by the Fleet hardware will be implemented by dividing the count among multiple instructions. Counting moves with multiple destinations will be implemented using instruction loops.

The Admiral compiler will avoid deadlock in the switch fabric by computing a conservative estimate of the amount of data traffic through each stage of the network and introducing flow control when the amount of queue space is exceeded. Instruction management in the switch fabric is more complicated, and the compiler will need to make use of data FIFOs to temporarily store instructions. This is necessary if the amount of space in an instruction FIFO runs out, to prevent other instructions from interfering with an instruction loop, to implement instruction loops that are larger than the size of an instruction FIFO, and possible for sequencing as well. Fleet provides a mechanism for transferring

```
foo: {
  ... // declarations
  // I1, I5
  a.out1 -> c.in1 ||
  {
    // I2, I8
    a.out2 -[2]-> e.in1 ;;
    // I3, I4, I6, I9
    b.out -> c.in2, e.in2 ;;
  } ||
  // I7, I10
  c.out -> e.in2
}
```

```
a.out1:
  take, sendto c.in1;       // I1
a.out2:
  [2] take, sendto e.in1;   // I2
b.out:
  take, sendto c.in2;       // I3
  sendto e.in2;             // I4
c.in1:
  accept;                   // I5
c.in2:
  accept;                   // I6
c.out:
  take, sendto e.in2;       // I7
e.in1:
  [2] accept;               // I8
e.in2:
  accept;                   // I9
  accept;                   // I10
```

Figure 10: An admiral block and its basic translation to dock instructions, before flow control or sequencing is added.

an instruction stored as data to its proper instruction FIFO, which allows Admiral to store instructions in data FIFOs or in memory.

5.2 Control Flow Graph

After translating the move statements in an Admiral program into Fleet dock instructions, the compiler builds a *control flow graph* representation of each named block in the program. The control flow graph contains a node for each dock instruction. For each sequencing operator in the named block, the graph contains a directed edge from each of the dock instructions corresponding to the statement to the left of the operator to each of the dock instructions corresponding to the statement to its right. Figure 11 shows the control flow graph for the program in Figure 10.

The control flow graph of a named block represents the sequencing requirements in the block. For every pair of instructions, if their is a path in the graph from the first to the second, then the first instruction must complete before the second can execute. If no such path exists, then the two instructions may run in parallel.

5.3 Instruction Dependency Graph

The compiler also builds an *instruction dependency graph* representation of each named block in the program. The instruction dependecny graph contains a node for each dock instruction. The graph contains a directed edge from one instruction to another if the latter depends on the former in some way, such that the latter can execute only after the former completes. For example, if an instruction depends on data produced by a second instruction, an edge would connect the second to the first. Another example of a dependency is if two instructions are to the same dock, and one is dispatched before the other. As discussed in §2.5, the first would arrive at the dock's instruction FIFO and therefor execute before the second. Figure 12 shows the control flow graph for the program in Figure 10.

The instruction dependency graph of a named block represents the actual sequence in which its instructions execute. As in the control flow graph, for each pair of instructions, if their is a path in the graph from the first to the second,

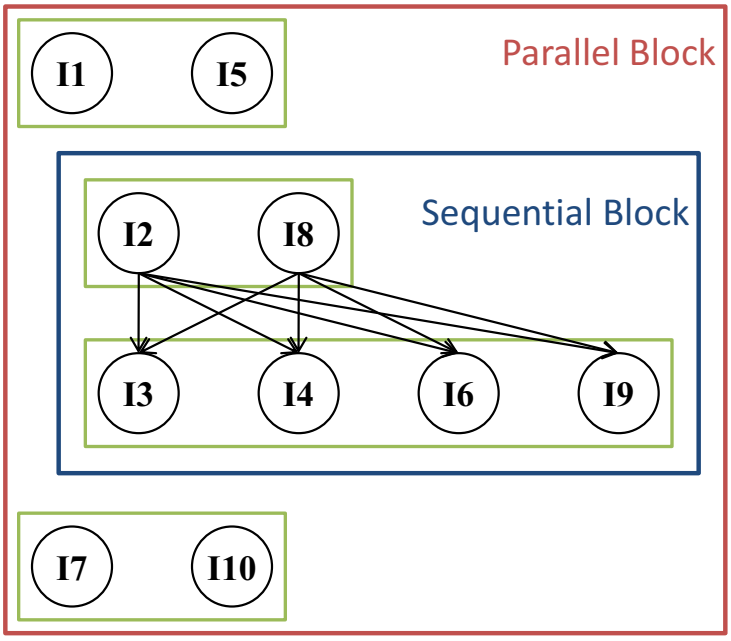

Figure 11: The control flow graph for the program in Figure 10.

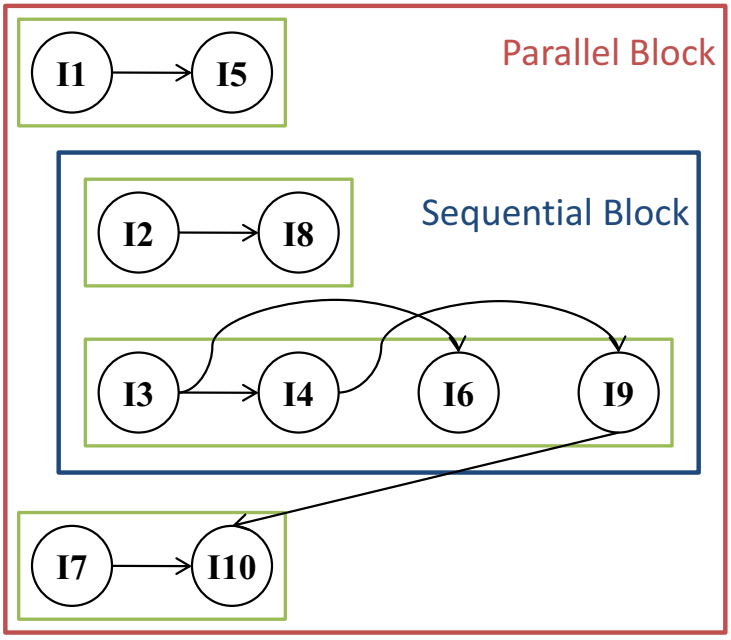

Figure 12: The instruction dependency graph for the program in Figure 10.

then the first instruction must complete before the second can execute. If no such path exists, then the two instructions can run in parallel. Thus, in order to implement the sequencing requirements in the control flow graph, the set of nodes reachable from each instruction in the control flow graph must be a subset of the nodes reachable from the same instruction in the instruction dependency graph. The compiler ensures this is the case by inserting new edges and nodes into the instruction dependency graph until the requirement is met. It is not yet clear what algorithm the compiler should use in doing this.

6 Discussion

There are a few open issues that the Admiral proposal does not address.

6.1 Functions

Most conventional programming languages include functions or subroutines in order to provide a clean interface between different parts of a program and to raise the level of abstraction. Virtual ships do accomplish this in Admiral, but the implicit setup and tear-down of pipelines when creating or destroying virtual ships make them ineffecient for infrequent operations. Thus, it may be necessary to add a function mechanism to Admiral. Since Admiral does not have an implicit stack, it is unclear how data would be passed between the function caller and the function itself.

6.2 Architecture Independence

In order for Admiral to be to Fleet as C is to sequential machines, it must provide an abstraction from the actual implementation of a Fleet architecture. In particular, an Admiral program should compile and run on any Fleet implementation, and the compiled program should take full advantage of the target Fleet.

Admiral attempts to provide some independence from implementation details discussed in §3. In particular, it is the job of the Admiral compiler to deal with various arbitrary limits and the topology of the switch fabric. However, Admiral does not achieve independence from the actual set of ships in a Fleet implementation.

It is possible for Admiral to provide independence from the *type* of ships in a Fleet implementation through a standard library of virtual ships. For example, if a Fleet implementation has no multiplier ship, a virtual multiplier can be provided that uses an adder ship. Considerale thought must be given to the details of the ships included in the standard library so that they can be efficiently emulated by many different Fleet implementations.

Unfortunately, Admiral does not have a mechanism for virtualizing the *number* of ships in a Fleet implementation. If a program is written to use a specific number of each type of ship, then it will not run on Fleet hardware with a fewer number of any type of ship. On the other hand, the program will not run any faster if the hardware has more ships than the program uses. Thus, Admiral programs are not scalable to the number of ships included in a Fleet implementation.

7 Conclusion

The Admiral language is a first attempt to design language around the features of the Fleet architecture. Admiral should simplify Fleet programming, is it removes the burden of dealing with many of Fleet's details from the programmer. At the same time, its proximity to the Fleet machine model should allow Admiral programs to perform nearly as well as Fleet assembly programs.

Admiral, however, is not the end-all of Fleet programming. As discussed in §6, Admiral programmers still need to know some details of a Fleet implementation in order to write code for it. Admiral may also be too low-level for

widespread use, as it requires the programmer to explicitly handle concurrency and communication. A higher-level, implicitly-parallel language with Admiral as its compilation target may end up being the ideal way to program Fleet.

References

[1] A. Megacz. Bypass Paths, August 2007. http://research.cs.berkeley.edu/class/fleet/docs/people/adam.megacz/am27.pdf.

[2] I. E. Sutherland. FLEET - A One-Instruction Computer, August 2005. http://research.cs.berkeley.edu/class/fleet/docs/people/ivan.e.sutherland/ies02-FLEET-A.Once.Instruction.Computer.pdf.