TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Fleet sorter: An Exercise into Programming FLEET

By:
Mike Holenderski

Supervisors:

Dr. Johan Lukkien (TU/e)
Dr. Igor Benko (Sun Microsystems)

Eindhoven, December 2006

Abstract

Today's shift in relative cost between computation and communication calls for a different approach to computing, moving away from the computation centric in the direction of communication centric approach.

Several alternatives to the traditional von Neumann architecture were proposed in the past, among them the dataflow machine. Fleet is an experiment into designing a modern processor architecture based on the static dataflow model. It employs asynchronous communication between the functional units, it is based on a distributed shared memory model, and can be implemented in hardware efficiently. Unlike other dataflow machines it is asymmetric and provides an instruction set comprised of a single move instruction.

In this Master Thesis we researched the Fleet architecture, focussing on the programability aspect. We selected a particular problem of merge sorting and presented a set of primitives which proved to be useful for designing and programming a sorter Fleet.

We evaluated the performance of the primitives by simulating the sorter Fleet in the ArchSim simulator. To gain understanding and support for the empirical results, we derived a theoretical performance model for the sorter Fleet.

Fleet tackles the program counter bottleneck by executing instructions as soon as their operands become available. In order to optimize the communication between the functional units, it provides a single move instruction, providing the programmer with explicit control over the communication. Unfortunately, our results indicate that Fleet is still haunted by the memory bottleneck. Since Fleet regards the memory just as any other ship, however, the bottleneck can be remedied easier than on a von Neumann machine.

Contents

Chapter 1

Introduction

The history of the modern computer started in 1946 with ENIAC. It was the first machine to use vacuum tubes instead of mechanical switches found in its predecessors. The vacuum tubes formed the functional units and were connected by simple copper wires. The cost of a vacuum tube was relatively high compared to the cost of the wiring needed to interconnect it with other tubes. It posed a limit on the number of tubes a computer could contain. The machine was programmed by manually routing wires. Then the von Neumann computer architecture emerged, allowing to reconfigure the computer by means of a program stored in the memory, rather than in the wiring. Still the relatively high cost of using the operation units called for operation centric program design, aimed at optimizing the usage of the computational units. Throughout the second half of the 20th century the vacuum tubes became replaced by transistors, allowing to fit more computational units per surface area.

Today the relative cost between computation and communication has changed. It is possible to fit more transistors on a chip than the designers can use, while the connections between the transistors occupy more chip area, consume more energy and are responsible for the delays. The challenge now lies not in squeezing as many transistors in the given area, but rather in transporting data efficiently between the functional units.

The shift in costs calls for a different approach to computing, moving away from the computation centric in the direction of communication centric approach. With this in mind, Fleet is an experiment into designing a modern processor architecture dictated by the current technology, rather than history.

1.1 Background

As computation became cheaper, researchers thought of ways to exploit it. In the 1970s some claimed that the complexity of individual computational units would soon reach their limits. In order to solve larger problems in a given time, they suggested using parallel computers containing several computation units. These would allow to execute instructions concurrently, allowing to perform more operations in a given time interval.

Fleet is not the first architecture to explore massive concurrency. In the early 1970's a lot of research focused on programming models involving parallel

processors. Some claimed that the von Neumann model was unsuitable for concurrent processing, due to the two bottlenecks: the program counter and the globally shared memory [2].

In a von Neumann machine the program is stored in a globally *shared memory*. Instructions are loaded from the memory to the processing unit, where they are matched with operands and executed. The *program counter* points to the next instruction to be executed, and thus governs the program execution.

The program counter bottleneck refers to the fact that an instruction is executed only when it is reached by the program counter. This allows to execute one instruction at a time. The memory bottleneck deals with the low throughput between the functional unit and the memory, due to the slow speed of memory relative to the functional unit.

According to the taxonomy of Flynn [13], the parallel processing models can be classified as Single Instruction Multiple Data(SIMD) or Multiple Instruction Multiple Data(MIMD). Examples of an SIMD are processor arrays or vector processors, where different processing units perform the same operation on different data. An MIMD, such as a multiprocessor computer, allows the processing elements to execute different instructions on different data.

In 1974 Dennis [11, 12] suggested a *dataflow* model [32, 18, 21], which falls into the MIMD category. In a dataflow machine, the computation is orchestrated by the availability of data, rather than by a centralized control of the program counter. The data stream determines the computation, rather than being a consequence of it.

Dennis thus tackled the program counter bottleneck. He introduced the graph representation of a dataflow program [11] and designed a dataflow machine which would execute it [12].

1.1.1 Dataflow Architecture

A dataflow graph represents a program which is intended to be executed by a dataflow machine. It is a directed graph, where nodes represent the primitive instructions and arcs define the data dependencies between the instructions. An example dataflow graph and the corresponding control flow program are shown in Figure 1.1.

Data travels along the arcs and is conveyed by tokens. The basic firing rule states that

- An instruction is enabled when all its operand arcs contain a token.

An enabled instruction may fire at any time. When it fires, it consumes the tokens from its operand arcs and produces a token on all its output arcs. The relation between the data conveyed by the input and output tokens is defined by the instruction.

The firing rule gets rid of the program counter bottleneck found in von Neumann machines, as an instruction can be executed as soon as its operands are available. This allows fine grained concurrency at the instruction level. Also, the execution becomes data driven, as control is shifted from the program counter to the data itself.

A dataflow architecture can be classified as *static* or *dynamic*, depending on the number of tokens an arc can accommodate. The difference in their behavior is reflected by their firing rules.

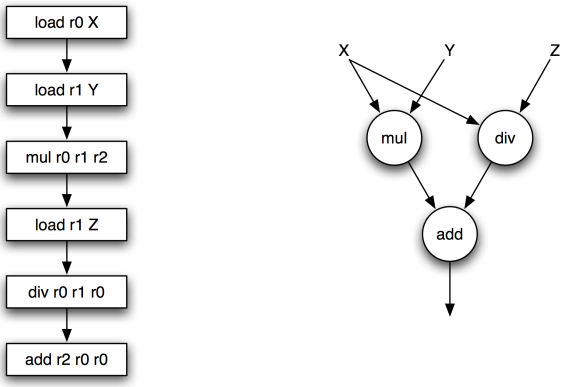


Figure 1.1: Controlflow vs. Dataflow representation of $X \cdot Y + \frac{X}{Z}$

Static Dataflow Machine

In static dataflow each arc can hold at most one token. This means that an instruction can be executed only when the output arc is empty. The firing rule thus becomes:

- An instruction is enabled when all its operand arcs contain a token *and* the output arc is empty.

The static dataflow machine was first suggested by Dennis [12] as an architecture for executing dataflow graphs [11]. The general structure of a static dataflow machine is shown in Figure 1.2.

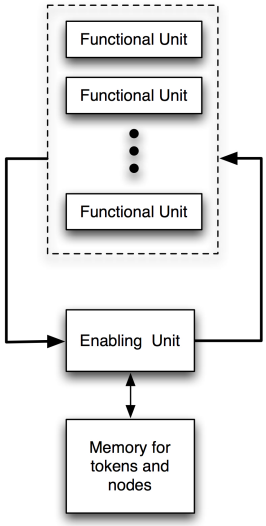


Figure 1.2: General structure of a dataflow machine.

The dataflow graph is stored in the memory in instruction cells. An instruction cell specifies the operation, the operands and the destination for the result, where a destination is the address of the operand the result of the current instruction is destined for.

A result from an instruction enters the *enabling unit*, which checks with the *memory* if the destination instruction has received all other operands. If so, the instruction is sent to one of the *functional units* and executed. If not, the incoming operand is stored in the memory in the corresponding instruction cell.

The data driven execution allows to execute instructions concurrently (provided a sufficient number of functional units) as soon as their operands are available and thus alleviates the program counter bottleneck. The memory bottleneck still remains, as the instructions and operands are stored in the shared memory and need to squeeze through the narrow path between memory and the enabling unit.

An example of a static dataflow architecture is the MIT dataflow machine [12]. It is *symmetric*, meaning the functional units are identical. It uses a *shared memory* for storing the program and the state, defined by the operand values in the instruction cells. Due to hardware restrictions, the number of operands in an instruction is limited to at most two. The execution is *asynchronous*, as the instructions can execute independently of one another. The asynchrony is supported by storing tokens in memory until all operands comprising an instruction have arrived before the instruction is fired.

Dynamic Dataflow Machine

Dynamic dataflow models arcs as bags which can contain several tokens. Each token is decorated with a tag. The firing rule becomes:

- An instruction is enabled when all its operand arcs contain tokens with the same tag.

When an instruction fires it consumes the tokens from its operand arcs and produces a token on all its output arcs with a new tag or the same tag as the input operands.

The dynamic dataflow can be regarded as an extension of the static dataflow, allowing multiple tokens in each arc, which in turn facilitates synchronizing access to shared resources. The additional functionality comes at a cost. The tags associated with tokens need to be managed by additional hardware and complicate the matching process.

The general structure of a dynamic dataflow machine is shown in Figure 1.3.

As there are possibly many instances of each instruction, one for every set of operands with the same tag, storing the destination address with each instruction instance is inefficient. It is more effective to split the enabling process into two phases [32]. First, when a token arrives from a functional unit the *matching unit* checks with the *token memory* whether all corresponding operands with the same tag are already waiting. If not, then the token is stored in the *token memory*. If all operands with the same tag are present, then the *fetch unit* retrieves the corresponding instruction from the *instruction memory* and passes it on to a *functional unit*.

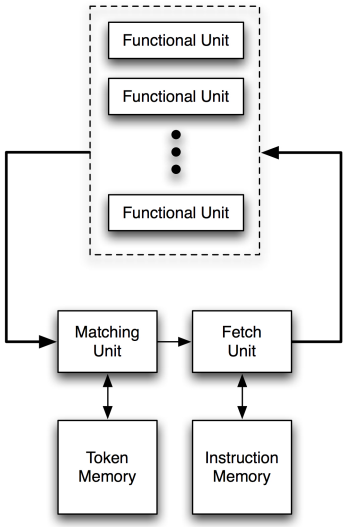


Figure 1.3: General structure of a dynamic dataflow machine.

The two most prominent examples of a dynamic dataflow machine are the Manchester Dataflow Machine [15, 16] and the MIT Tagged Token Dataflow Machine [1]. Both are *symmetric* , *asynchronous* and based on the *shared memory* model.

Differences

The main difference between the static and dynamic dataflow lies in the way they handle reentrant code, such as multiple loop iterations, or multiple calls to a subroutine. For example, the subgraph defining a loop is executed multiple times. Static dataflow allows only a single token to occupy an arc, meaning only instructions from a single iteration can be executed at a time, and thus concurrency is limited to partially overlapping loop iterations. Dynamic dataflow allows multiple tokens on an arc, each equipped with a tag. The tag can represent tokens belonging to the same iteration. This allows to concurrently execute several iterations of a loop, while leaving the process of matching the tokens with the same tag (i.e. from the same iteration) to the machine.

1.2 From Dataflow to Fleet

Since its advent, the von Neumann architecture has dominated the computer world. However, it has two major problems: the program counter and the memory bottlenecks.

The dataflow architecture remedies the program counter bottleneck through data driven program execution. There are, however, several drawbacks to dataflow:

- The data driven execution comes at a cost. The enabling mechanism requires large associative memory to store the operand tokens. The match-

ing mechanism of tagged tokens in dynamic dataflow is especially costly, as most implementations apply hashing techniques, which are typically slower than associative memories in static dataflow.

- Fine grained concurrency has its benefits in parallel execution, but it requires large overhead for enforcing sequential execution. Attempts were made to tackle the overhead by balancing fine and coarse grained concurrency.

  One can think of the von Neumann and dataflow architectures as two extremes on the axis representing the grain of computation, with von Neumann being efficient in the coarse grain computation and dataflow on the fine grain. An ideal architecture will be somewhere in between [21, 18, 17]. The hybrid approach, for example, suggests to execute groups of interdependent instructions in sequence on a von Neumann machine, but schedule the execution of the sequences in a dataflow manner.

- Since the operands of instructions are matched in the instruction memory, the von Neumann memory bottleneck still remains.

In the eary 1990s research in pure dataflow suddenly stopped, mainly due to the overhead necessary to compensate for the fine grain concurrency inherent to dataflow and the difficulty to design efficient hardware implementations to support it [18].

Fleet [25, 24, 28, 23, 27, 26, 29, 30, 5] is an attempt to design an efficient computer, which addresses the issues of fine grained concurrency experienced by dataflow, in the context of todays technology. The fact that computation is becoming cheaper than communication calls for a new approach to designing hardware, but also a new approach to computation in general.

The Fleet model resembles the *static dataflow model*. It employs *asynchronous* communication between the functional units, it is based on a *distributed shared memory* model, and can be implemented in hardware efficiently.

However, unlike most static dataflow architectures researched in the past [12, 1], it is *asymmetric*, where the functional units are not necessarily identical.

While other dataflow machines operate on the level of operations such as addition or multiplication, the Fleet instruction set contains only a single *move* instruction, which gives the programmer explicit control over the communication between the functional units.

Like other dataflow architectures, Fleet remedies the von Neumann program counter bottleneck by data driven execution, but it goes further and tackles several drawbacks of dataflow.

- Fleet introduces a *switch fabric* which is responsible for token matching. Unlike the (conceptually) unbounded instruction memory in dataflow, the switch fabric has a limited capacity. On the one hand it is simple and can be implemented in hardware efficiently. On the other hand, some of the control over the scheduling of execution is moved to the programmer, who is responsible for coordinating which instructions enter the switch fabric.

- The overhead of sequential execution due to the very fine grained concurrency is tackled by moving parts of sequential computation inside the functional units. The asymmetry of functional units allows to define complex units, which perform higher level operations.

- Fleet addresses the von Neumann memory bottleneck by matching operands with instructions in a fully distributed and concurrent *switch fabric* [10].

Having only a single instruction has far reaching consequences. Since one programs Fleet solely by specifying data movement between the functional units, the computation becomes an implicit byproduct of the communication. This is a dramatic shift in the approach to computing, and consequently programming.

1.3 Goal of this thesis

The goal of this thesis is to investigate the programming aspect of Fleet. We will demonstrate the features of Fleet, identify several key issues encountered while programming Fleet, and define a set of programming primitives which address these issues. We chose to do this research using an example problem: sorting. As Fleet is in its initial stage of research, this will be one of the first programs written for Fleet. We will provide feedback and advice for future research into Fleet.

Chapter 2

Fleet

The goal of Fleet is to research an efficient computer architecture which will provide insight into utilizing the available concurrency. Its design is dictated by the current technology rather than building on top of the technology used in the past. It falls into the category of static dataflow.

The core observations guiding the design of Fleet can be summarized as [25]:

- Simplicity reduces the cost at all levels, from design to programming.
- Communication consumes most resources (chip area, energy, time).
- Concurrency is available due to the low cost of logic and is waiting to be exploited.

Petri Nets We will use petri nets to describe the behavior of different components comprising the Fleet architecture. A petri net consists of *places*, *transitions* and *directed arcs*. Arcs are only allowed between places and transitions. Places with arcs ending at transition t are called *input places* of t, and places with arcs starting at transition t are called *output places* of t.

Places may contain tokens. A transition is *enabled* when all its input places contain at least one token. An enabled transition may *fire*, in which case it consumes one token from each input place and produces one token in each output place.

We consider *1-bounded* petri nets, where all places can contain at most one token. As a consequence a transition is enabled when all input places contain a token *and* the output places are empty.

2.1 The Architecture

In general, a *computation* deals with computing an answer to a given question. The question is the input and the answer is the output of the computation. As it is not feasible to design a computation for every question from scratch (we simply have too many questions), people thought of decomposing the original computation into smaller computations, in such a way that their results can be combined to form the answer to our original question, and so that they can be reused for answering other questions.

If we consider computation in the context of current *computers*, the smaller computations correspond to the *functional units*, while the coordination of the computation is specified by the *program*, composed of instructions provided by the *instruction set* of the computer.

The functional units in a Fleet computer are referred to as *ships*. Fleet provides an instruction set containing a single *move instruction*, which allows to specify data movement between the ships, and a *switch fabric*, which executes the movement.

2.1.1 The Ship

Ships are the functional units of Fleet and are responsible for modifying the data flowing through the Fleet. All devices, such as memory, registers, or other functional units, are regarded as ships. In fact a ship can be an arbitrary subroutine.

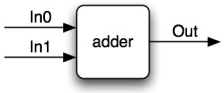


Figure 2.1: An adder ship.

An example of a ship is shown in Figure 2.1. It depicts an adder ship with two inputs $In0$ and $In1$ and one output Out. Figure 2.2 depicts a petri net describing its behavior.

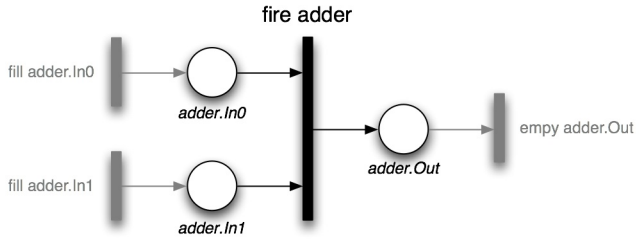


Figure 2.2: A petri net describing the behavior of an adder ship.

A token represents a single data. When there is a token at both places *adder.In0* and *adder.In1*, the adder will consume them and produce a token containing their sum in *adder.Out*. The gray transitions represent the rest of the Fleet around the adder ship, which provides the input data to the ship and consumes the output data. For now we abstract from the details and model it with simple transitions.

The petri net in Figure 2.2 models an adder without internal buffering; the output is produced directly after the inputs are consumed. An example of a ship experiencing independent behavior of the input and output interface is shown in Figure 2.3.

It is a FIFO ship with one input *In* and one output *Out*, implementing a First In First Out queue. A petri net modeling the behavior of a two stage fifo

Figure 2.3: A fifo ship.

ship is shown in Figure 2.4.

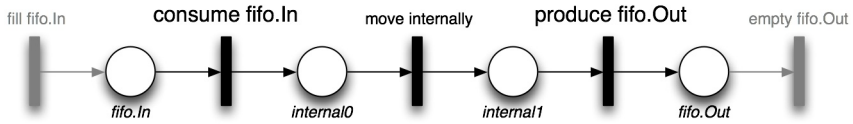


Figure 2.4: A petri net describing the behavior of a two stage fifo ship.

The *consume fifo.In* and *produce fifo.Out* transitions model the input and output interfaces of the fifo ship. The gray transitions represent the environment. A two stage fifo ship can buffer at most two data elements and it returns the data elements in the same order as it received them. The behavior of the input interface is decoupled from the output interface due to the internal buffering.

A ship may have any number of inputs and outputs. An input or output is active if and only if it contains a data. When enough inputs are active the ship is activated and consumes the data from these inputs. A ship can produce a data on an output at any time after consuming the inputs, but only when the corresponding output does not contain any value. Fleet poses no restriction on the time between consuming inputs and producing outputs. The relation between the inputs and outputs defines the function of the ship.

A particular fleet may contain several instances of a particular ship kind, which can be exploited for concurrency. The set of ships defines the functionality of a particular Fleet.

2.1.2 The move Instruction

Fleet aims at optimizing the communication between the functional units and therefore puts it under direct control of the programmer. It provides an instruction set containing a single *move* instruction.

A move instruction

$$\text{move } x \rightarrow y$$

moves the data from the *source* x to the *destination* y. The sources are the ship outputs (as they produce data) and the destinations are the ship inputs (as they consume data). For example

$$\text{move adder.Out} \rightarrow \text{fifo.In}$$

will move data from the output terminal *Out* of the adder ship to the input terminal *In* of the fifo ship. A move "begins" when data is available at the

source and "ends" when the data is consumed by the ship at the destination. Each ship in a Fleet is assigned a unique name, so that there is no ambiguity in referring to sources and destinations when a Fleet contains more than one instance of a particular ship. Hence adder.Out is a unique source address and fifo.In is a unique destination address.

In contrast to the traditional von Neumann computer and other dataflow machines discussed in Section 1.1, a Fleet program specifies data movements between the functional units, rather than specifying the operations to be executed by the functional units on the data. For example, the standard assembler operation ADD x y z is represented in Fleet by three move instructions:

move $x \rightarrow$ adder.In0
move $y \rightarrow$ adder.In1
move adder.Out $\rightarrow z$

where locations x and y are outputs of some ships and location z is an input of a ship. Note that in von Neumann architecture the ADD operation is implemented as a set of three data moves

- move data from register x to the first operand of the functional unit
- move data from register y to the second operand of the functional unit
- move data from the output of the functional unit to register z.

Hence the data moves comprising the ADD operation are still executed, however, they are hidden from the programmer. Fleet exposes the dataflow and puts it under direct control of the programmer. It can potentially result in an increase in performance in situations where some of the intermediate data moves can be skipped, e.g. the result of the addition can be sent directly to the input of a subtractor ship, instead of storing it first in a register.

The RISC approach of a single-instruction instruction set also allows for more flexibility in the functional units in terms of their interfaces and behavior. A ship is not bound by the number of inputs it can consume, nor by when and how many outputs it can produce.

An important consequence of the move centric control is that computation becomes an implicit byproduct of communication. The user only controls the movement of data, moving data between the outputs and inputs of ships. A ship consumes the inputs and generates new outputs, thus modifying the data stream.

2.1.3 The Switch Fabric

Data is moved between the outputs and inputs of the ships by the *switch fabric*.

Fleet is silent about the design and implementation of the fabric. In our particular Fleet we consider a *horn-funnel switch fabric*. It consists of five parts: source funnel, trunk, destination horn and the instruction horn and instruction pool, as shown in Figure 2.5.

Figure 2.6 depicts the journey of an instruction. The instruction pool contains instructions which are ready to execute. It issues these instructions in any order it pleases. An instruction specifies a source and destination. It leaves

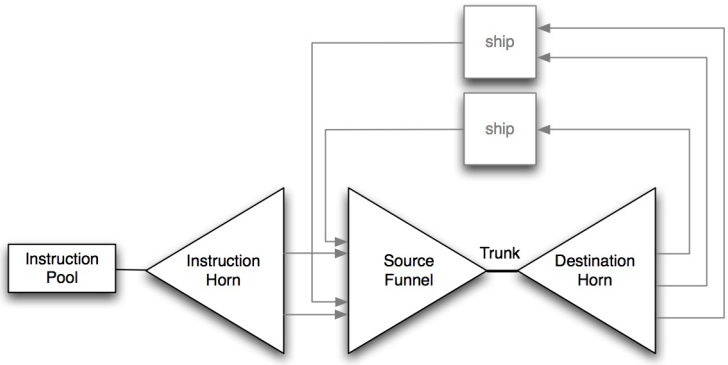


Figure 2.5: A general view of a Fleet.

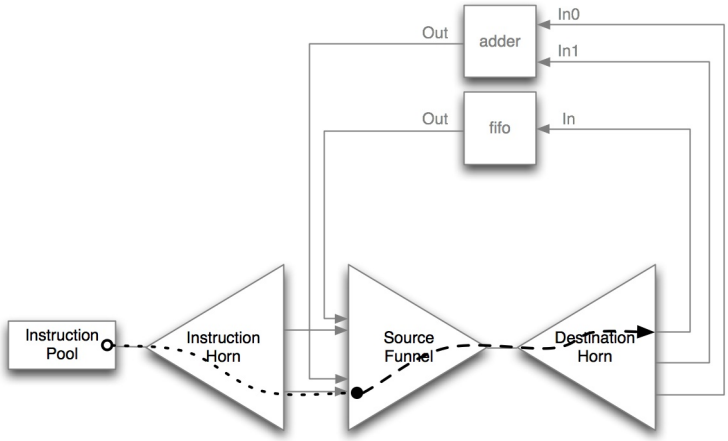


Figure 2.6: A Fleet with an adder, fifo and fetch ship. The black arrow depicts the journey of the instruction: move adder.Out → fifo.In

the instruction pool and enters the instruction horn. The horn delivers the instruction to the source specified in the instruction. At the source it will wait for a data. When the data arrives, the instruction grabs it and moves through the source funnel and trunk to the destination horn, which then delivers the instruction together with the data to the destination specified in the instruction. The instruction's journey, and its life, ends upon delivering the data to the destination.

Asynchrony

Fleet is silent about the design and implementation of the switch fabric. Different designs of the switch fabric are possible, which will optimize different parameters, such as latency, throughput, chip area or energy consumption [25]. However, Fleet does require the switch fabric to be *asynchronous*, meaning an instruction will wait at the source for the data from a ship, and it will wait at

the destination for the ship to be ready to accept the delivered data.

Asynchronous communication requires buffering for storing the data which is not synchronized yet. In dataflow machines [12, 1] the operands of partially filled instructions are stored in the memory. They remain in the memory until all operands have arrived and the instruction can be executed.

Buffering In Fleet the switch fabric has buffers *at least* at two places: the *sources* at the source funnel, where instructions and data from a ship output wait for each other, and at the *destinations* at the destination horn, where instructions wait for the ship input to accept the transported data. The memory bottleneck found in other dataflow machines is remedied by fully distributing the instruction buffers.

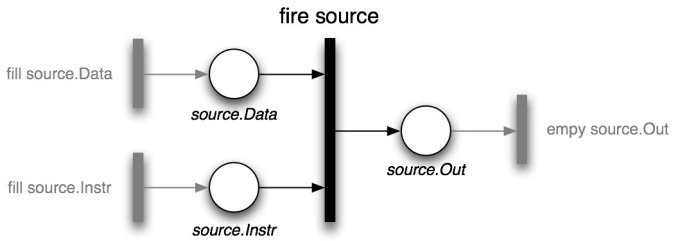


Figure 2.7: A petri net depicting the behavior of a source.

Each output of a ship is connected to a source. The source is responsible for merging the data with an instruction which will transport the data to its destination. The behavior of the sources at the source funnel is shown in Figure 2.7. The *source.Data* is the output place of the corresponding ship. Example of a source together with the corresponding ship output is shown in Figure 2.8.

Note the different meaning of the tokens in different places. The token in *source.Data* represents a data on the output of a ship, the token in *source.Instr* represents an instruction intended to move data from this source, and the token in *source.Out* represents the instruction merged with the data.

Routing In this thesis we consider the *horn-funnel switch fabric*. The horns and funnel are each composed of a network of one stage fifos, as shown in Figure 2.9. They are arranged in binary trees with the sources and destinations forming the leaves. Each leaf is assigned a unique address, whose bits determine the path through the corresponding tree.

- An instruction leaves the instruction pool and enters the instruction horn.
- The black squares are able to store a single value. They represent the wires, a latch and some logic needed to move data.They are connected by *selectors* (gray triangles), which route the instruction to the appropriate source. A selector consumes the most significant bit of the source address in the instruction and forwards the instruction to one of its branches accordingly. The behavior of a selector component can be modeled by a petri net shown in Figure 2.10.

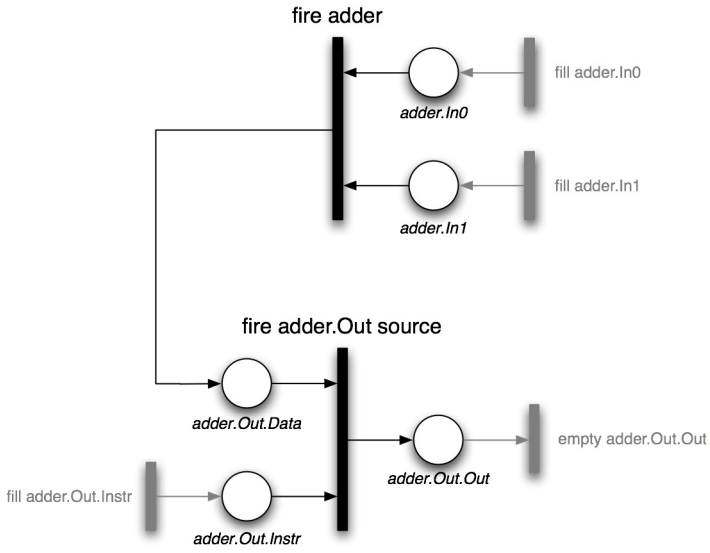


Figure 2.8: A petri net depicting the behavior of an adder ship together with the source.

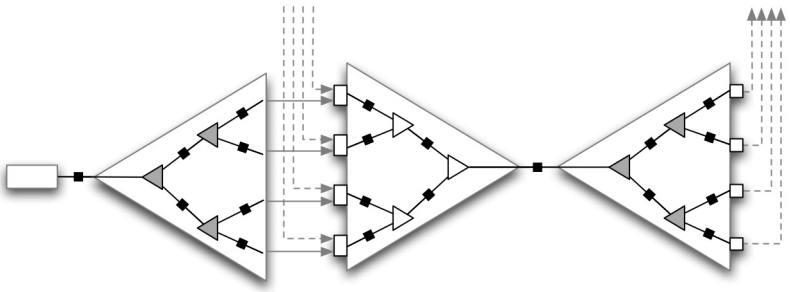


Figure 2.9: The internals of the horn-funnel switch fabric

A token represents an instruction. An instruction token in the *input* place can be consumed by only one of the transitions. Usually in a petri net the choice is nondeterministic. Here we assume that choice depends on the most significant bit of the address in the instruction.

- The data from the outputs of the ships meet with the instructions from the instruction horn at the *sources* (white rectangles) at the source funnel. These insert the data into the instruction and pass it on to the funnel. The source will wait for both the data and the instruction before passing them on to the source funnel. The behavior of a source was shown in Figure 2.7.
- The source funnel is comprised of a network of *arbiters* (white triangles), which merge the instruction stream through the funnel. In case two instructions arrive at the same time the arbiter randomly decides which one

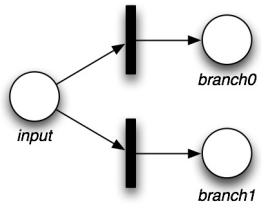


Figure 2.10: The behavior of a selector inside a horn.

to pass through first. A petri net depicting the behavior of an arbiter is shown in Figure 2.11.

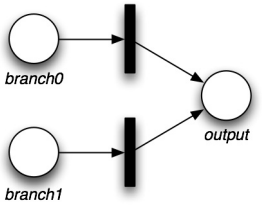


Figure 2.11: The behavior of an arbiter inside a funnel.

Since a place can contain at most one token, only one transition can be enabled at a time. If both branches contain a token, one transition will fire and the other one will have to wait for the output place to be empty again. The choice between the transitions is nondeterministic.

- The trunk connects the source funnel with the destination horn.
- The destination horn routes the instruction to the appropriate destination similarly to the instruction horn, but based on the bits specifying the destination address. At the destination the data is extracted form the instruction (white rectangle) and passed on to the ship.

Congestion The switch fabric does not overwrite values in the buffers. A data or instruction is passed to the next buffer only when the next buffer is empty. If an instruction cannot deliver the data to the ship, then it will remain at the destination in the destination horn. Other instructions with the same destination will pile up behind it, thus clogging up the horn and the trunk, until no other instructions can reach their destination, leading to a deadlock. Similarly an instruction waiting for the data at the source funnel may congest the instruction horn, preventing new instructions from entering the switch fabric.

Flexibility One important consequence of the asynchrony in the switch fabric is that it offers flexibility in timings of the ships. This benefit can help improve the performance of the ships and reduce their design and running cost. A ship can, for example, have a synchronous implementation, as long as it can

communicate in an asynchronous way with the switch fabric. The flexibility, however, comes at a cost of making the computation less predictable.

2.1.4 Move Enhancements

The simple move instruction is extended with several features intended to simplify the programming of the machine.

Literals

Fleet allows to insert literals straight into the switch fabric, as an alternative to having a ship produce the literal value on its output and moving it from the corresponding source to the destination. A literal can be of any data type supported by Fleet. The source in a move instruction can be replaced by a literal surrounded by parenthesis. E.g.

$$\text{move } (1) \rightarrow y$$

will move the literal 1 directly to the destination y, bypassing the instruction horn and the source funnel. For this purpose a direct connection is added to the switch fabric, connecting the instruction pool with the trunk, as shown in Figure 2.12.

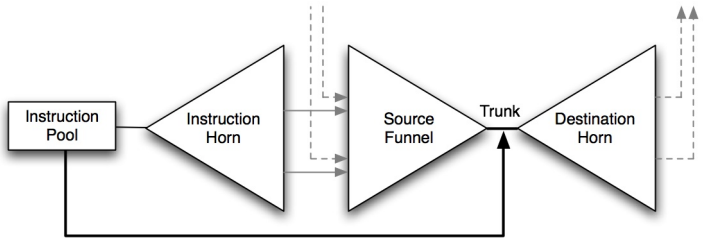


Figure 2.12: Support for literals.

Out-of-Band data

Every Fleet data type is extended with Out-Of-Band (OOB) values. They represent special boundary values, e.g. for integers there could be one OOB data representing minus infinity, one representing plus infinity and another representing "not an integer" value. These OOB values can be used by ships to handle simple boundary conditions.

Every data type has a special OOB value called LAST. It designates the last element in a sequence of values. For example a ship producing a sequence of memory addresses can designate the end of the sequence by a LAST value.

The OOB extension allows to encode control of the dataflow inside data itself, reinforcing the data driven nature of Fleet.

Multiple Destinations

The desintation field in a move instruction can specify multiple destinations. E.g.

$$\text{move } x \rightarrow y_1, y_2$$

will move a copy of the data from source x to destinations y_1 and y_2. A multiple destination move instruction is handled in the switch fabric by the source x, which will dispatch two single moves with destinations y_1 and y_2, each merged with a copy of the data at x.

The source field can be a literal, in which case a copy of the literal will be moved to y_1 and y_2. A multiple destination literal move is converted to two single moves in the instruction pool.

It is important to remember that the multiple destination move is converted to several single partial moves *at the source*, rather than in the instruction pool. This guarantees that no other move instruction*with the same source* will execute in between the partial moves. We refer to them as partial, since they are created at the source and have only a destination.

However, it could very well happen that an instruction with the same destination, but a different source (e.g. a move $z \rightarrow y_2$ in the example above), will complete between the partial moves. Therefore multiple destinations are semi-atomic instructions.

Standing Moves

Sometimes we may need to move a whole data stream between two ships.

For example imagine we would like to read an array of data elements from the memory. This could be done with two ships, a memory ship and a stride ship, where the stride would generate consecutive addresses of the array elements and send these to the address interface of the memory ship. The memory ship would then read the elements of the array and send these to some other ship.

Rather than issuing multiple copies of the same move instruction between the stride and the memory ship, it can be more efficient to issue one *standing* move instruction,

$$\text{move[standing] } x \rightarrow y$$

Such an instruction will reside at the source and move the incoming data to the destination until the instruction is disassembled. This is where the Out-Of-Band data values are usefull. Upon an incoming LAST data the source will convert the standing move to a single move, and move the LAST data to the destination.

One can regard a standing move as wiring up the source and destination for the time of their communication. After the ships are done conversing, the wiring is removed.

There is also a *counting* variant of the standing move, where the number of moves is bounded by some constant k,

$$\text{move}[k] \; x \rightarrow y$$

A *counting move* will move at most k copies of data from x to y. It may be disassembled before reaching k by an incoming LAST data.

2.1.5 Code bags

Move instructions are grouped into *code bags*. A code bag is identified by a *code bag descriptor* and contains a bag of move instructions.

CBD {
    move $a \rightarrow x$
    move $b \rightarrow y$
    move $c \rightarrow z$
}

The instructions within a code bag execute concurrently, hence their order cannot be specified simply by the sequence inside the code bag. This means that a conflict between instructions sharing the same source or destination may arise, when these are issued concurrently.

A Fleet *program* consists of several code bags, and always contains a special *initial code bag* which is executed first and bootstraps the rest of the program.

2.1.6 Program Execution

Every Fleet contains a special *fetch* ship. The fetch ship has one input on which it expects a code bag descriptor.

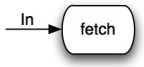


Figure 2.13: Fetch ship.

When a code bag descriptor arrives, the fetch ship will fetch the corresponding code bag and dump its content into the *instruction pool*, which is responsible for buffering and issuing instructions to be executed. The newly fetched code bag will execute concurrently with the instructions already residing in the instruction pool. One can picture the fetch process as grabbing a small bag of instructions and dumping its content into a big bag representing the instruction pool. The big bag is then shaken and the next instruction is blindly taken out and executed.

Every program has one initial code bag which is fetched and released into the instruction pool when loading the program. This code bag is then responsible for loading other code bags.

The fetch mechanism allows to load the next piece of the program by simply moving a code bag descriptor to the fetch ship. The code bag descriptor itself can be produced by another ship, or included as a literal in a move instruction. This elegant fetch mechanism nicely follows the simplicity guideline. Treating code bag descriptors as data enhances the data-driven nature of Fleet.

2.1.7 Example: Fibonacci Fleet

We will demonstrate the working of Fleet with a simple example computing the Fibonacci sequence, given by $F_n = F_{n-1} + F_{n-2}$ with $F_0 = F_1 = 1$

Our Fibonacci Fleet is shown in Figure 2.14. It contains three ships: an adder, a fifo and the fetch ship. The adder has two inputs and one output. When both inputs are active it will consume them and produce their sum on the output. The fifo ship is a one stage fifo with one input and one output. It will be used to store the F_{n-2} term between iterations.

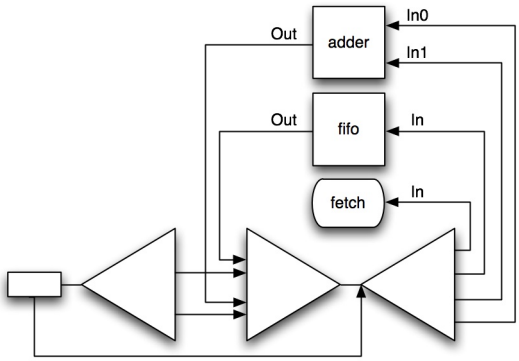


Figure 2.14: The Fibonacci Fleet

The Fleet program is shown below.

```
init codebag START {
    move (1) → fifo.In, adder.In0, adder.In1
    move[standing] fifo.Out → adder.In1
    move (ADD) → fetcher.In
}
codebag ADD {
    move adder.Out → adder.In0, fifo.In
    move (ADD) → fetcher.In
}
```

The behavior of the Fibonacci Fleet is defined by the petri net in Figure 2.15. It is a composition of the petri nets[1] defining the behavior of the separate components discussed above.

The execution of the move instructions are depicted in Figure 2.16, where we abstract from the switch fabric and indicate only he data movement.

Initially the START code bag is released into the instruction pool. Note that the three instructions can execute in any order. The first instruction will insert a 1 into the fifo, setting up $F_0 = 1$ and load the adder with F_0 and F_1. The second instruction moves the output out the fifo to the second input of the adder. Note that it is a standing move, meaning it will remain at the fifo.Out and move all incoming data to adder.In1 until it is disassembled[2]. The third instruction moves the code bag descriptor ADD to the fetch ship, which fetches the instructions contained in the ADD code bag and releases them into the instruction pool.

[1] We composed petri nets similar to composing snippets in [8]

[2] Our simple Fibonacci Fleet does not terminate and the standing move is never disassembled.

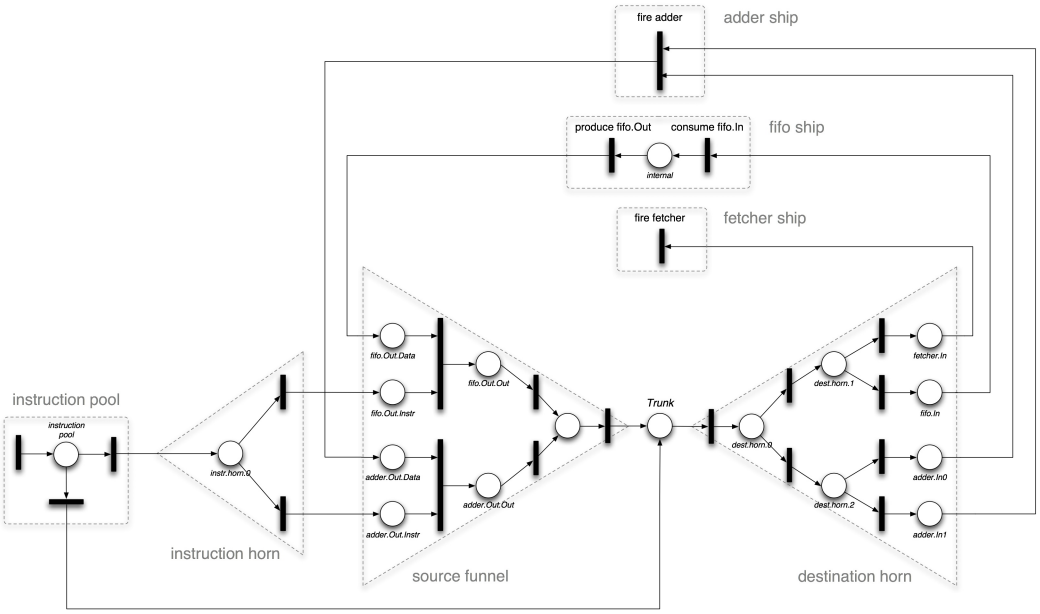


Figure 2.15: A model of the behavior of the Fibonacci Fleet

The first instruction in the ADD code bag recycles the F_n term, by setting up the next addition iteration and storing it for the following iteration in the fifo.

This simple Fleet example does not terminate and keeps computing the next Fibonacci number in the sequence.

Since the instructions can execute in any order, it could happen that

move (ADD) → fetcher.In

would be the only instruction to be continuously executed. It would load the two instructions comprising the ADD code bag into the instruction pool, and always execute the latter one. This would result in a live lock, where the number of instructions

move adder.Out → adder.In0, fifo.In

in the instruction pool would only increase.

A deadlock is not possible in this fleet. A deadlock occurs when there is a data in the switch fabric, which can not be delivered to its destination. It can occur in two ways:

- No instruction moving data from the source to a destination is released into the instruction pool, e.g. because it is not specified in the program at all, or because the code bag containing it cannot be reached.
- A move instruction is released into the instruction pool and then into the switch fabric, but it cannot progress due to congestion.

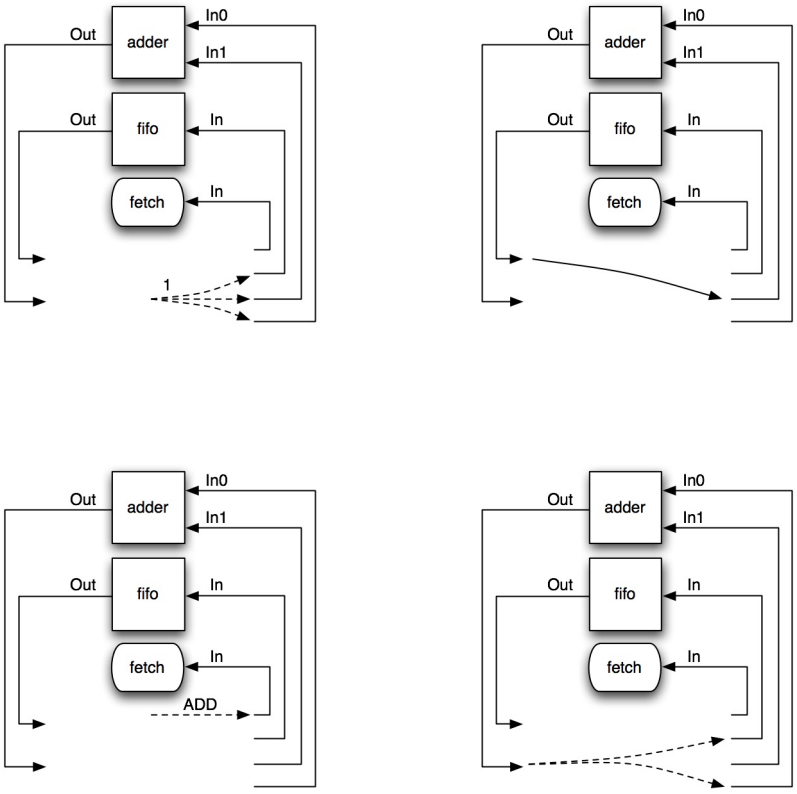


Figure 2.16: Execution of the Fibonacci Fleet

In our Fibonacci Fleet the adder produces data on its output only after the inputs were supplied. The inputs, however, depend on the output of the adder. Since the Fleet is initialized with a single set of inputs to the adder, when a data is produced on the adder.Out there is no other data from previous iterations clogging up the switch fabric. In every iteration two data items are produced (a copy of the data form adder.Out is sent to adder.In0 and fifo.In) and two data items are consumed (by the adder from the adder.Out and fifo.Out), hence the number of data items in the switch fabric is at most 3 (together with the initialization).

2.2 Concurrency

Fleet is inherently concurrent. From the programming perspective it means that things can happen in nondeterministic order.

- Since the switch fabric may be concurrent, the instructions issued by the instruction pool into the switch fabric may be completed by the switch fabric in any order, subject only to the availability of data on the ship outputs and the ability to deliver data to the ship inputs.

- The fetch ship may fetch instructions from different memory locations, some may be closer in fast caches, some may be further away in the main memory. This means that instructions from the second half of a code bag may be released into the instruction pool earlier than the instructions from the first half.
- The fetch ship may start fetching the next code bag before it has finished fetching the previous code bag. This means that instructions from different code bags may also execute in any order.

Therefore when a code bag is fetched, we can picture that the instructions are released in the instruction pool and mixed with instructions already residing there.

This means that different code bags can execute concurrently, but also instructions inside a code bag can execute in any order. A well designed program which is resilient to the side effects of this concurrency can potentially exploit the fine grained parallelism. However, we found that it is difficult to design such programs due to two problems we may encounter: *conflicting sources* and *conflicting destinations*.

Conflicting sources occur when at least two instructions are released into the instruction pool which share the same source,

move $a \rightarrow x$
move $a \rightarrow y$

Since we cannot predict the order of their execution, we cannot predict whether the first data from a is going to be sent to x or to y.

For example, given a two-stage fifo ship containing two numbers 2 and 5, and a subtractor ship, we would like to subtract the first two numbers from a fifo and then feed the result back to the fifo. If we were to release the following instructions concurrently

move fifo.Out → subtractor.In0
move fifo.Out → subtractor.In1
move subtractor.Out → fifo.In

then the result fed back to the fifo would be either a 3 or a -3, depending on the order in which the first two instructions would execute.

Conflicting Destinations occur when at least two instructions are released into the instruction pool which share the same destination,

move $x \rightarrow a$
move $y \rightarrow a$

Again, since we cannot predict the order of their execution, we cannot predict whether a is going to receive the first data from x or from y.

For example, let our Fleet contain an accumulator ship with one input and one output. The accumulator ship consumes integer numbers on the input until it receives a LAST value, upon which it returns on the output the cumulative sum of all received numbers prior to LAST. Initially the accumulator is set to 0. If we were to release the following instructions concurrently

move (2) → accumulator.In
move (5) → accumulator.In
move (LAST) → accumulator.In

the instruction moving the LAST literal to the accumulator could execute before the other two instructions feeding the accumulator. If the LAST move will execute first, the accumulator will return 0. The literals 2 and 5 will be consumed by the accumulator without producing a result.

If the LAST move will execute after the other two instructions, then the accumulator will return 7.

2.3 Sequence

There are two ways of controlling sequence in Fleet: depending on the dataflow through ships and using the code bag fetching mechanism.

Ships fire and produce outputs only after receiving and consuming enough inputs. For example, given three concurrent instructions

move fifo.Out → adder.In0
move fifo.Out → adder.In1
move adder.Out → fifo.In

the last move cannot complete before the adder ship will produce data on the Out output. The adder will produce an output only after it receives data on both of it's inputs. This means that the first two instructions must execute before the last one.

The code bag fetch mechanism provides another way to sequence program execution. The program can be split into several code bags, e.g. A, B and C. Code bag C will be released only after the code bags A and B have completed. The question is how do we know that a code bag has completed. There is no way of telling from simply counting the instruction in the instruction pool, since after an instruction leaves the instruction pool and enters the switch fabric, its execution is determined only by data availability at the sources and destinations. Alternative ways need to be found, possibly involving other ships in the Fleet.

2.4 Implementation

The key feature of data flow architecture is the fine grain parallelism. However, in the 1970's it was soon realised that the overhead due to the fine grain parallelism could not be supported by the contemporary hardware.

Fleet aims at providing an efficient hardware implementation of the static dataflow architecture. FleetZero [10] was an experiment conducted at Sun Microsystems into implementing an asynchronous switch fabric, based on GasP circuits [22, 31, 14]. A GasP circuit is a network of alternating PLACE and PATH circuits, where the PATHs control the dataflow through the PLACEs. The state in a PLACE is stored in a single wire, similar to the single track handshaking by Peeters and van Berkel [20]. It avoids the extra acknowledgment cycle found in static dataflow implementations from the past.

Chapter 3

Sorting Fleet

The aim of this thesis is to gain insight into programming Fleet. Programming in general deals with reusing given hardware components to solve a given computation problem. We decided to solve the sorting problem on Fleet. Since at this point there is no fixed Fleet configuration, we will also investigate what ships such a Fleet could contain. We will thus tackle the problem from two sides, trying to answer the questions: what do we require from the Fleet hardware to program it and what can the Fleet hardware provide.

In this chapter we will briefly introduce merge sort in Section 3.1, describe the general picture of our sorting Fleet in Section 3.2 followed by a bottom up design of the Fleet with Section 3.3 discussing synchronization to solve the source and destination conflicts and Section 3.4 describing the data driven control. Section 3.8 sums up the discussion and states our general programming primitives for solving the encountered issues.

3.1 Merge Sort

The merge sort algorithm was invented by John von Neumann in 1945 [19, 9]. We consider a variant of merge sort which can merge C streams. The input array is distributed over the C streams. C is called the *sorter capacity* and refers to the length of the input array which it can completely sort in one iteration. For example, a single sorting element in Figure 3.3 has capacity 2, while the network of three such sorting elements shown in Figure 3.3 has capacity 4.

Our variant of the merge sort algorithm for $C = 4$ is summarized by the following pseudo code in Algorithm 1. The operation of this algorithm is visualized in Figure 3.1.

The algorithm consists of two functions with the following specifications:

function mergesort:
arguments: *input*: array of numbers
return: array containing elements from *input* in ascending order

function merge:
arguments: $l0, l1, l2, l3$: list of numbers
return: a single list containing elements from $l0, l1, l2, l3$ in ascending order

Algorithm 1 Merge Sort for $C = 4$

function mergesort(*input*: array of numbers):
  $N := length(input)$;
  $chunkSize := 1$;
  $chunkCount := N$;
  $nextChunk := 0$;
  $merged := input$;
  while $1 < chunkCount$ **do**
    $notmerged := merged$;
    $merged := \emptyset$;
    while $nextChunk < chunkCount$ **do**
      $chunk0 := notmerged[offset...offset + chunkSize)$;
      $chunk1 := notmerged[offset + chunkSize...offset + 2 \cdot chunkSize)$;
      $chunk2 := notmerged[offset + 2 \cdot chunkSize...offset + 3 \cdot chunkSize)$;
      $chunk3 := notmerged[offset + 3 \cdot chunkSize...offset + 4 \cdot chunkSize)$;
      $merged := merged :: \text{merge}(chunk0, chunk1, chunk2, chunk3)$;
      $offset := offset + 4 \cdot chunkSize$;
      $nextChunk := nextChunk + 4$;
    end while
    $chunkSize := chunkSize + 4$;
    $chunkCount := chunkCount/4$;
  end while
  return *merged*

function merge($l0, l1, l2, l3$: list of numbers):
  if $empty(l0) \wedge empty(l1) \wedge empty(l2) \wedge empty(l3)$ **then**
    return
  else if $head(l0) = \min(head(l0), head(l1), head(l2), head(l3))$ **then**
    return $head(l0) :: \text{merge}(tail(l0), l1, l2, l3)$
  else if $head(l1) = \min(head(l0), head(l1), head(l2), head(l3))$ **then**
    return $head(l1) :: \text{merge}(l0, tail(l1), l2, l3)$
  else if $head(l2) = \min(head(l0), head(l1), head(l2), head(l3))$ **then**
    return $head(l2) :: \text{merge}(l0, l1, tail(l2), l3)$
  else if $head(l3) = \min(head(l0), head(l1), head(l2), head(l3))$ **then**
    return $head(l3) :: \text{merge}(l0, l1, l2, tail(l3))$
  end if

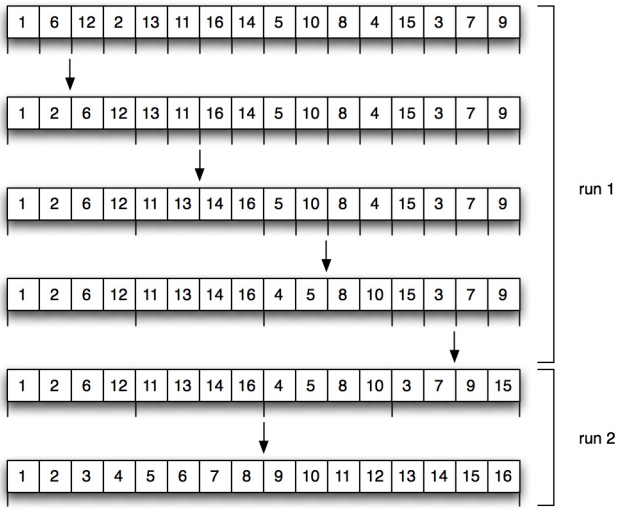


Figure 3.1: Merge sort involving a sorter with capacity 4.

The $head(l : list)$ function returns the first element of list l if l is not empty, and ∞ otherwise. The $\min(x_1, x_2, x_3, x_4 : number)$ function treats a $LAST$ data as $-\infty$. All other functions have the common meaning.

Let us assume that the length of the input array is N. If $N \leq C$ then the input can be sorted in one go, by simply feeding it through the sorter. If $N > C$ then sorting requires several *runs* through the input, where each run produces longer sorted *chunks*, until there is only a single chunk left (see Figure 3.1). Initially there are N chunks, each containing a single data element.

3.2 Merge Sort Fleet

An example of a possible sorting Fleet is shown in Figure 3.2.

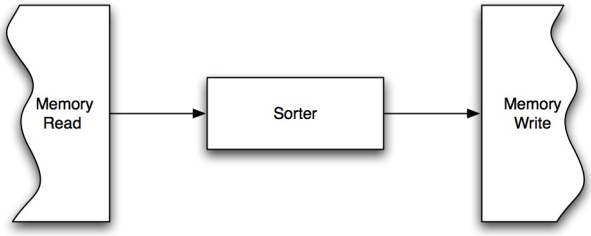


Figure 3.2: A sorter Fleet with a single sorter ship.

If we had a sorter ship which would read in an input array from the memory and write it back to the memory in correct order, then the sorter Fleet would contain three ships: a *MemoryRead* ship providing a read controller for the memory, a *Sorter* ship and a *MemoryWrite* ship providing the write controller

to the memory. The sorter ship could be a general purpose processor which would run a sorting program. Such a Fleet, however, has a few drawbacks.

Fixed storage space inside of ships If we assume the comparison of two numbers to be the basic step, sorting an input array of length N requires $O(N \log N)$ steps. This means that if the sorting ship shown in Figure 3.2 was to sort an input array in one pass, it would have to have $O(N \log N)$ complexity, where the ship complexity is defined by the number of registers and functional units, or simply the number of gates. Since we assume ships in a Fleet (with the exception of MemoryRead and MemoryWrite) have a constant complexity, such a single "mighty" sorter ship is not possible.

Concurrency in the Switch Fabric We can relax the one pass requirement and let the sorter perform several runs through the input array in the memory while sorting it. The sorter ship now will be able to request a chunk of the input array, sort it and write back to the memory. Also, while a single sequential sorter ship has to alternate reading and writing of data to and from the memory and the sorting, several sequential sorters allow to overlap memory access with computation.

However, a Fleet containing a single sorter ship cannot take advantage of the concurrency offered by the switch fabric. If we introduce several additional sorters, we could distribute parts of the sorting problem between them, sort these parts concurrently and merge them into a single sorted sequence.

There are limitations on the number of sorters which can be used efficiently, depending on the speed of the memory relative to the speed of the sorters and the bandwidth of the Switch Fabric. If the memory is the bottleneck of the system, then increasing the number of processors will not speed things up. We can only speed up the Fleet by adding more sorters if the sorters or the communication are slower than the memory.

Chip area vs. generality We must keep in mind that the purpose of Fleet is to allow to control the movement of data between different ships and thus to reuse ships for different computation problems. We could therefore design a Fleet which would contain many specific purpose ships, such as a sorter ship, which can sort a sequence of any length, internally dividing the sorting process into runs and managing the reading and writing of data from and to the memory.

However, ships are meant to be implemented as hardware components on a chip. As a chip has a finite surface area, it can fit only a finite number of gates, and thus a finite number of ships. To allow Fleet solving general problems, we need to select what ships a Fleet should contain. A complicated sorter ship as suggested above will occupy a relatively large chip area while allowing to solve only sorting problems. Ideally we would like to have ships which are both general and occupy little space. It is difficult to measure generality and we will gain insight into which ships are useful while designing Fleets to solve different problems. For now let us decide on a simple merge sort ship, which merges two sorted streams into one sorted stream.

We can split the sorter into two parts: a merge sorting ship and a coordinator ship, responsible for managing the runs and loading data from the memory through the sorter and back to the memory. Following the same reasoning, a

coordinator ship able to manage runs of a merge sorting algorithm is complex but not very general. We therefore split it into several other more general ships, such as a Stride managing the memory read address, a stride managing the write address, and several arithmetic ships, such as an adder and a shifter, to manage the runs.

3.2.1 General Overview

Imagine a simple sorting ship with a limited sorting capability, e.g. a fixed number of input elements the sorting ship can handle at a time. Our goal is to design and program a Fleet which will combine several of these simple ships to sort inputs of larger size.

We consider a sorting ship with two inputs and one output. Each input accepts a sequence of integer numbers sorted in ascending order. These two sequences are merged in ascending order and produced on the output.

Our Fleet contains several such ships. We can combine these as shown in Figure 3.3 to allow sorting four sequences simultaneously. We can then employ the standard merge sort technique and iterate several times to sort an input array of any size.

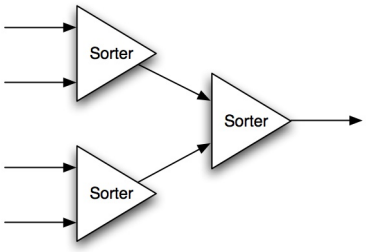


Figure 3.3: Three sorters combined to form a bigger sorter

The sorter ships alone are not able to access data in the memory. We therefore introduce auxiliary ships controlling the dataflow between the memory and the sorters. A general overview of our sorting Fleet is shown in Figure 3.4.

The memory control is distributed among two ships, representing the read and write interfaces. Data is read from the memory, channeled through the sorter and written back to the memory.

A cloud represents a collection of ships comprising a *channel*. We will use the term *read channel* to refer to the ships handling the stream of data from the memory feeding a single sorter input. Similarly we will use *write channel* to refer to the ships handling the writing of the data from the output of the last sorter to the memory.

Within a run, chunks are assigned to read channels and then merge-sorted to form longer chunks. Merging chunks consists of reading the data elements comprising the chunk to the right channel, passing through the sorters and writing back to the memory. Thus the sorting process involves loading runs, each run involves loading chunks, and each chunk involves loading data, as shown in Figure 3.5.

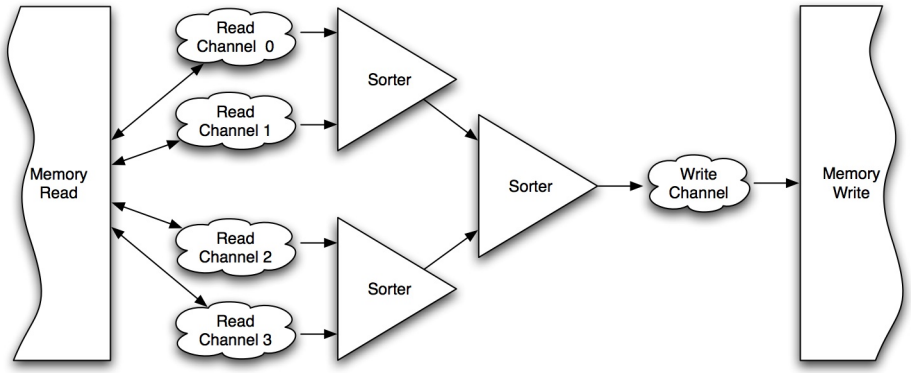


Figure 3.4: A general picture of the dataflow through the sorter Fleet

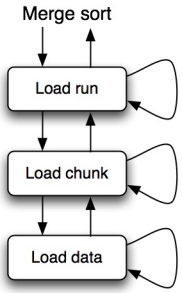


Figure 3.5: Merge process.

At each level the decision is made whether to continue or to return to the higher level. For example, after loading a data item from a chunk, the read channel will decide whether to load the next data from the same chunk or whether to start loading data from the next chunk. The write channel is responsible for deciding when to load the next run.

3.2.2 Memory Organization

The input array is residing in the memory. We pass it through the fleet several times in such a way that in the end the memory contains the elements from the input array in ascending order.

We assume that each chunk ends with a LAST *Out-Of-Band (OOB)* data, and that the first element in the memory at offset 0 contains the number of elements to be sorted (excluding the OOB data), as shown in Figure 3.6. Initially all chunks have size 1.

With respect to the memory management, in-place merge sort, where the same memory space is used for reading and writing data, is difficult [19].

During a run data is read from the memory, merged and written back to the memory. Each channel is responsible for reading its own data and it may proceed at its own pace, depending on whether the read values are smaller or

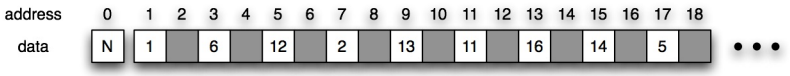


Figure 3.6: Format of the input residing in the memory, with gray rectangles representing OOB data.

greater than the values in other channels. This will create irregular gaps in the input array. The partially sorted output sequence is written to the memory in consecutive chunks. In order to avoid conflicts between reading and writing chunks, the data is stored in two consecutive blocks, the *reading block* and the *writing block*, starting at an offset R and W, respectively.

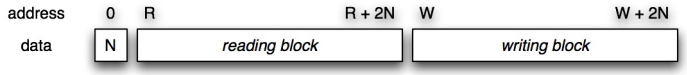


Figure 3.7: Memory is divided into read and write blocks

We avoid overwriting elements which have not yet been read by reading from the reading block and writing to the writing block. At the end of the run the reading block is empty and the writing block contains the partially sorted data from the reading block. If still another run is necessary to completely sort the data, the two blocks are swapped, i.e. in the next run the writing block becomes the reading block and the reading block becomes the writing block. Swapping the blocks is accomplished simply by swapping the offsets R and W. Initially $R = 1$ and $W = 2N + 1$[1].

3.3 Synchronization at the Memory

A *read channel* represents the collections of ships which read and stream data from the memory into a specific input of a sorter ship. A channel is responsible for reading its data from the memory whenever the corresponding sorter input is ready to accept new data. The channel is data driven, i.e. the flow of data through the channel is controlled by the data itself.

Let us zoom in on the interface between the channels and the memory read interface. We will design the memory read interface step by step.

3.3.1 Memory Read Interface

A simple memory read ship is depicted in Figure 3.8. It has one input *Address* and one output *Data*. When it receives an address on the input, it will fetch the data from the corresponding memory location and return it on the output.

[1]The $2N$ is due to the LAST data items terminating each chunk. Initially there are N chunks each containing 1 data item.

In our sorter Fleet the MemoryRead ship is servicing the read channels. If there is a single channel requesting data from the memory ship, this interface might work well, as shown in Figure 3.8.

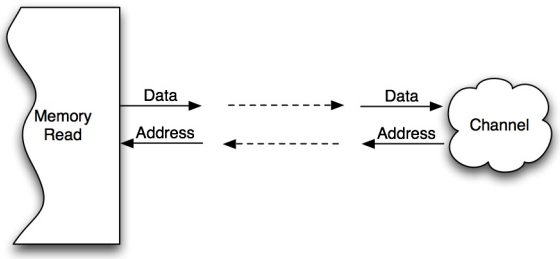


Figure 3.8: A single channel requesting data from the memory read ship.

The dashed lines represent two move instructions moving data between the Channel and the MemoryRead:

move Channel.Address → MemoryRead.Address
move MemoryRead.Data → Channel.Data

The first move instruction will supply the memory ship with an address and the second will move the requested data back to the channel.

However, in our Fleet there are several channels which request data from the memory, as shown in Figure 3.9.

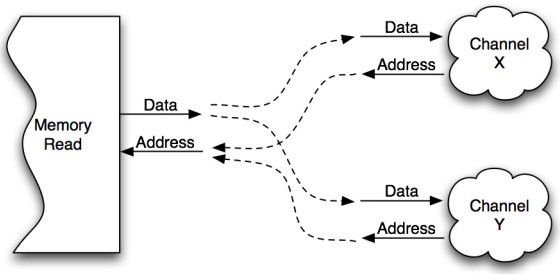


Figure 3.9: Two channels requesting data from the memory read ship.

If two channels try to access the memory ship at the same time, each will dispatch code bags containing move instructions supplying its address to the memory ship and transporting the data back:

move ChannelX.Address → MemoryRead.Address
move MemoryRead.Data → ChannelX.Data

move ChannelY.Address → MemoryRead.Address
move MemoryRead.Data → ChannelY.Data

This simple example exposes two major problems encountered when programming Fleet trying to maintain concurrency: *conflicting sources* and *conflicting destinations*. They are a manifestation of the Fleet platform offering only very fine grained atomic operations (the move instructions).

3.3.2 Conflicting Sources

Let us first consider the move instructions transporting data from the memory to the channels.

After the memory ship is presented with an address it will fetch the corresponding data from the memory and publish it on the Data output. Since there are two channels requesting data from the memory, two instructions will be executing concurrently:

move MemoryRead.Data → ChannelX.Data
move MemoryRead.Data → ChannelY.Data

Since they are concurrent, they may execute in any order. As a consequence, ChannelY may receive the data which was requested by ChannelX, and vice versa.

We will remedy this problem by preventing two instructions with the source MemoryRead.Data from entering the instruction pool concurrently. For this purpose we extend the memory read ship and the channels with a code bag descriptor interface, as shown in Figure 3.10.

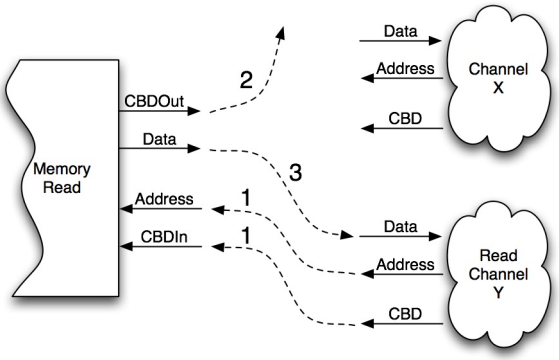


Figure 3.10: The code bag descriptor interface.

An address supplied to the memory ship will be accompanied by a code bag descriptor[2]. The code bag descriptor will point to a code bag containing the move instruction transporting the requested data back to the appropriate channel. We therefore wrap the conflicting move instructions in two code bags

```
MOVE_DATA_CHANNEL_X {
    move MemoryRead.Data → ChannelX.Data
}
```

[2]Section 3.3.3 describes how to supply the address and the code bag descriptor in an atomic step.

```
MOVE_DATA_CHANNEL_Y {
    move MemoryRead.Data → ChannelY.Data
}
```

The MemoryRead ship will store internally the code bag descriptor together with each request and once the data is fetched it will publish the data with the corresponding code bag descriptor. The code bag descriptor will be moved to the fetch ship, which will fetch and issue the appropriate move instruction transporting the data to its desired channel.

3.3.3 Conflicting Destinations

Let us take a look what happens when two channels try to request data from the memory concurrently.

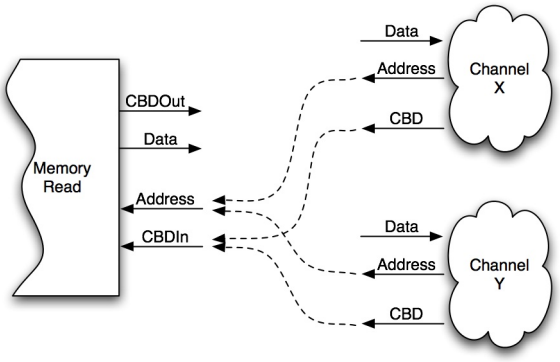


Figure 3.11: Two cbd enabled channels requesting data from the MemoryRead ship.

Both channels will request data from the memory by dispatching two moves each.

```
move ChannelX.Address → MemoryRead.Address
move ChannelX.CBD → MemoryRead.CBDIn

move ChannelY.Address → MemoryRead.Address
move ChannelY.CBD → MemoryRead.CBDIn
```

Since the instructions may execute in any order, it could happen that the ChannelX address could end up paired with ChannelY code bag descriptor and consequently the read data would be sent to the wrong channel, defying the purpose of the code bag descriptor interface.

To remedy this we introduce a pipeline interface[3], as shown in Figure 3.12. A channel can produce data on its *Address* and *CBD* outputs only after it receives a token on the *Next* input.

[3]Our pipeline interface is similar to the interface suggested by Ivan Sutherland [25]. It differs in the way it handles LAST data. While Sutherland's output pipeline will consume an incoming LAST token without producing an output, our interface will react to it as if it was a valid token and produce an output.

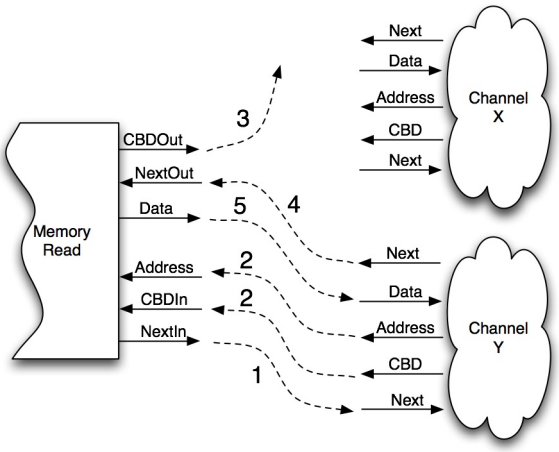


Figure 3.12: The memory and channels with a pipeline interface

When both channels try to request data from the memory simultaneously, both will dispatch a code bag containing a move instruction trying to claim the token from the memory ship:

move MemoryRead.NextIn → ChannelX.Next
move MemoryRead.NextIn → ChannelY.Next

However, due to arbitration in the source funnel only one of the move instructions will be able to obtain it. As a result, only one channel will produce data on its address and code bag descriptor interface. As a move instruction can only complete when the data on the source is available, the move instructions of only one channel will execute, preventing the conflict between the channels.

When the *MemoryRead* ship has successfully received a request from one channel and is ready to accept a new request, it will produce a token on *NextIn*. This token will be then claimed by the other channel, which will then submit its request.

The complete *MemoryRead* ship is described below. We will describe the behavior of ships as concurrent programs, using the following syntax.

- $identifier = program$ attaches an *identifier* to a *program*

A *program* can be composed of the following atomic steps

- $terminal?variable$ meaning that data is read from the *terminal* and stored in the *variable*
- $terminal!variable$ meaning that the value stored in the *variable* is written to the *terminal*
- $variable1, variable2 := data1, data2$ being a concurrent assignment of $data1$ and $data2$ to $variable1$ and $variable2$

The allowed compositions are

- a ; b denoting sequential composition, meaning a is followed by b
- $a \parallel b$ denoting parallel composition, meaning a and b may execute in any order or concurrently
- **if** ... **then** ... **else** ... **end if** for branching
- **while** ... **do** ... **end while** for conditional iteration
- (a)* for unconditional iteration, meaning a can execute 0 or more times, depending on the environment

Further there are two predicates

- $valid(a)$ returning true if a is valid
- $oob(a)$ returning true if a is OOB

and two constants

- $token_{valid}$ being a valid token
- $token_{oob}$ being an OOB token

MemoryRead

The *MemoryRead* ship is a memory controller allowing to read data from the memory. It can pipeline several read requests and process them in any order. It is equipped with a pipeline interface and a cbd interface for submitting a code bag descriptor, which will be published at the cbd output when the corresponding operation has completed. It specifies what to do with the data retrieved from the memory.

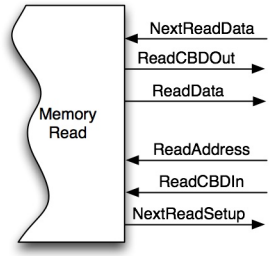


Figure 3.13: The MemoryRead ship

The behavior of the *MemoryRead* ship is captured by the following concurrent program:

initially i = 0;

$MemoryRead =$
$i := i + 1 \mod bufferSize$;
$(ReadAddress?address[i] \parallel ReadCBDIn?cbdIn[i])$;
if $valid(address[i]) \wedge address[i]$ within memory range **then**
  $(NextReadSetup!token_{valid}$; $MemoryRead) \parallel$
  $(ReadCBDOut!cbdIn[i]$; $NextReadData?token$;
  $ReadData!$ data at memory address $address[i])$
else
  $NextReadSetup!token_{oob}$; $MemoryRead$
end if

□

Note that $address[i]$ and $cbdIn[i]$ each point to a location in a buffer with size set to *bufferSize*. The buffers are introduced to allow the *MemoryRead* ship to pipeline several read requests before returning the fetched data. To simplify the program we assume that data can be read from an input to a buffer location only after the previously buffered data was written to an output, e.g. $ReadAddress?address[i]$ and $ReadData!address[i]$ for the same i must alternate.

3.4 Data Driven Control in Read Channels

The data-triggered execution model of Fleet, where ships are activated when enough of their inputs are active, suggests a data driven algorithm. We would like to create a Fleet which will decide when to execute certain operations, and also which operations to execute, based on the data itself.

In this section we will demonstrate the data driven nature of our Fleet sorter. Not only is the execution of a Fleet program data driven (due to the firing rule of the ships), but also our algorithm itself. In contrast to the traditional merge sort with centralized control, where a processor or a process is dedicated to coordinating the dataflow, our Fleet sorter distributes the control among the data itself.

The sorter Fleet uses the validity information of every data element (i.e. whether it is valid or OOB) read from the memory do decide what to do next, whether to load the next data element or start loading the next chunk.

Let us take a look inside the read channels clouds in Figure 3.4. Each channel is assigned two ships: a *ChannelStride* and a *ChannelCBD*. These keep loading the next data element from the memory to the sorter until the end of the chunk, designated by an OOB data. At this point the channel requests the next chunk.

The ChannelStride is an instance of the *SimpleStride* ship. It produces the address for the next data in the chunk currently read by the channel.

SimpleStride

A *SimpleStride* generates consecutive integer numbers. After being set up with a start value, it produces consecutive integers until it is set to a different value or overflows.

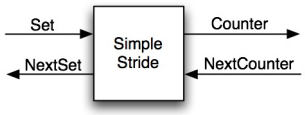


Figure 3.14: The SimpleStride ship.

The behavior of the *SimpleStride* ship is captured by the following concurrent program:

$SimpleStride =$
$Set?data$;
$counter := data$;
if $valid(data)$ **then**
  $NextSet!token_{valid}$;
  $(NextCounter?token$; $Counter!counter$; $counter := counter + 1)^*$;
else
  $NextSet!token_{oob}$;
  $NextCounter?token$; $Counter!counter$;
end if
$SimpleStride$

□

When a channel is assigned a new chunk the *ChannelStride* is initialized with the address of the first data element in the chunk. On any subsequent token on the *NextCounter*, the stride will generate the address of the next data in the chunk and increase the counter by 1. This will go on until a new chunk is loaded.

When to load the next chunk is decided by the *ChannelCBD* ship. It is an instance of the *OOBSelector* ship.

OOBSelector

The *OOBSelector* is a sibling of the boolean selector ship. It accepts three inputs, two branches and input which selects between the two. It selects the branch based on the validity of the data on the select input, i.e. whether it is valid or OOB, and forwards the data from the corresponding branch input to the output.

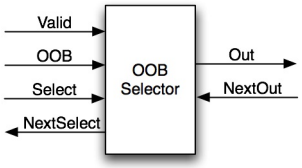


Figure 3.15: The OOBSelector ship.

The behavior of the *OOBSelector* ship is captured by the following concurrent program:

$OOBSelector =$
$(Valid?validBranch \parallel OOB?oobBranch \parallel Select?select \parallel NextOut?token)$;
if $valid(select)$ **then**
  $(Out!validBranch \parallel NextSelect!token_{valid})$;
else
  $(Out!oobBranch \parallel NextSelect!token_{valid})$;
end if
$OOBSelector$

□

The *ChannelCBD* takes its name from the fact that the two branch inputs are always code bag descriptors. It selects between the two based on the validity of the last data read by the channel. Depending on whether the last data read from the memory was valid or not, it will produce the code bag descriptor from the *Valid* or *OOB* input. The code bag descriptor is then sent to the fetch ship. The code bag descriptor on the *Valid* input will load the next data and the one on the *OOB* input will load the next chunk[4].

To complete the channel we need the input of the sorter ship. Sorters are also equipped with pipeline interfaces, as shown in Figure 3.16.

Sorter

The *Sorter* ship merges two ascending sequences on inputs $In0$ and $in1$ into one ascending sequence published on output Out. An OOB on either input designates the end of the corresponding sequence.

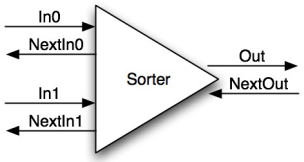


Figure 3.16: The Sorter ship with a pipeline interface.

The behavior of the *Sorter* ship is captured by the following concurrent program:

$Sorter = (In0?buffer0 \parallel In1?buffer1 \parallel NextOut?token)$; $Merge$

$ReadIn0 = In0?buffer0$; $Merge$

$ReadIn1 = In1?buffer1$; $Merge$

[4]An OOB data designates the end of a chunk.

$Merge =$
$NextOut?token$;
if $valid(buffer0) \wedge valid(buffer1) \wedge buffer0 \leq buffer1$ **then**
  $Out!buffer0$; $NextIn0!token_{valid}$; $ReadIn0$
else if $valid(buffer0) \wedge valid(buffer1) \wedge buffer0 > buffer1$ **then**
  $Out!buffer1$; $NextIn1!token_{valid}$; $ReadIn1$
else if $valid(buffer0) \wedge oob(buffer1)$ **then**
  $Out!buffer0$; $NextIn0!token_{valid}$; $ReadIn0$
else if $oob(buffer0) \wedge valid(buffer1)$ **then**
  $Out!buffer1$; $NextIn1!token_{valid}$; $ReadIn1$
else if $oob(buffer0) \wedge oob(buffer1)$ **then**
  $Out!token_{oob}$; $NextIn0!token_{valid}$; $Sorter$
end if

□

An overview of the working of a channel is shown in Figure 3.17. The large numbers indicate the sequence of events.

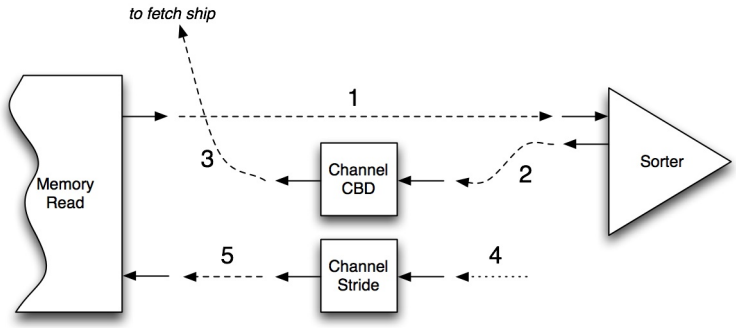


Figure 3.17: Channel deciding whether to load next data or next chunk

Data is read from the memory and sent to the sorter input (step 1). The pipeline interface attached to the input generates a token acknowledging the data. If the data was valid then a valid token is produced and when the data was OOB then an OOB token is produced. The token is sent to the ChannelCBD (step 2), which is based on the validity of the token dispatches the appropriate code bag descriptor to the fetch ship (step 3). The fetch ship fetches the code bag, which will either move the address of the next data from ChannelStride to MemoryRead (step 5), or first set the ChannelStride to the offset address of the next chunk (step 4) and then move the address from ChannelStride to MemoryRead (step 5).

We will now go into the details of Figure 3.17. First we zoom in on steps 2 and 3, and describe the decision process whether to load the next data element or start a new chunk, depicted in Figure 3.18.

The large numbers indicate the sequence of events. Dashed arrows are single move instructions. A dashed arrow with text on top indicates a literal move instruction, where the text specifies the literal. An arrow with an empty arrow-

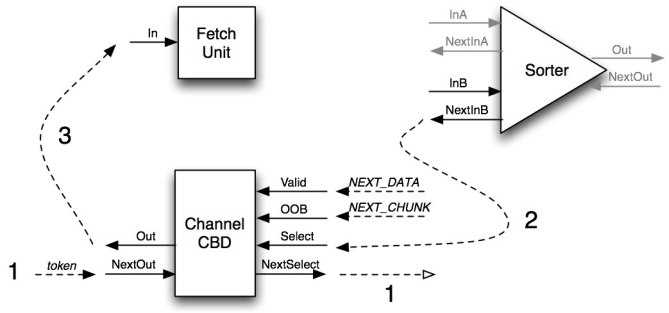


Figure 3.18: Channel deciding whether to load next data or next chunk

head represents a move to the *bitBucket*[5].

The sequence is enforced by the firing of the ships and the use of pipelines. Note that the handshaking introduces sequential behavior and thus limits the concurrency in the interaction between the memory and channel. Therefore our sorter Fleet contains several channels, which can operate concurrently.

The token acknowledging the last data received by the sorter is moved to the ChannelCBD. The token is valid if the input data was valid, in which case the next data must be loaded (see Section 3.4.1). If it is invalid it means that the last data read from the memory was the end marker for the chunk, hence the next chunk must be loaded (see Section 3.4.2).

3.4.1 Loading Next Data Element

When the sorter acnowledges a valid data, it sends a valid token to the SchannelCBD (depicted in figure 3.18 by arrow 2). The token then releases the *NEXT_DATA* codebag, shown in Figure 3.19.

```
codebag NEXT_DATA_TO_CHANNEL_1 {
    move Channel1CBD.NextSelect -> bitBucket
    move (MOVE_DATA_TO_CHANNEL_1) -> Channel1CBD.Valid
    move (EMPTY) -> Channel1CBD.OOB

    move (token) -> Channel1CBD.Select

    move Channel1CBD.Out -> Memory.ReadCBDIn
    move Channel1Stride.Counter -> Memory.ReadAddress

    // claim the token
    move Memory.NextReadSetup -> Channel1Stride.NextCounter,Channel1CBD.NextOut
}
```

Figure 3.19: *NEXT_DATA* code bag loading the next data to channel 1.

The process of loading data is shown in Figure 3.20.

Move instructions comprising steps 1-3 are released by the *NEXT_DATA* code bag, shown in Figure 3.21. Instruction 4 is a standing move set up during

[5]The *bitBucket* is a special destination address in the switch fabric which acts as a sink for any incoming data.

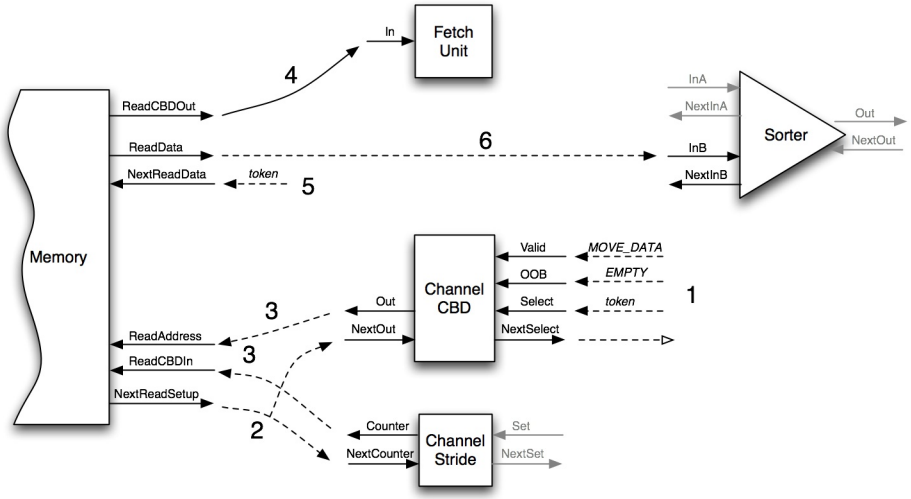


Figure 3.20: Channel loading next data item

the initialization (see Section 3.6). The *MOVE_DATA* code bag contains the moves 5 and 6.

```
codebag MOVE_DATA_TO_CHANNEL_1 {
    // read data from memory
    move (token) -> Memory.NextReadData
    move Memory.ReadData -> Sorter0.InB

    // setup ChannelCBD for next iteration
    move (NEXT_DATA_TO_CHANNEL_1) -> Channel1CBD.Valid
    move (NEXT_CHUNK_TO_CHANNEL_1) -> Channel1CBD.OOB
    move Channel1CBD.NextSelect -> Channel1CBD.NextOut
    move Channel1CBD.Out -> fetcher.In

    // get the token from the sorter input
    move Sorter0.NextInB -> Channel1CBD.Select
}
```

Figure 3.21: *MOVE_DATA* code bag moving a data from memory to channel 1.

First the *ChannelCBD* is setup with the *MOVE_DATA* code bag descriptor. Here ChannelCBD acts as a synchronization mechanism releasing *MOVE_DATA* only when it is the channels turn to request data from the memory. Since the *Select* token is always valid, the *EMPTY* code bag descriptor never gets released. It is only a stub needed to complete the inputs to *ChannelCBD*.

In step 2 the token from *NextReadSetup* is claimed by an instruction with multiple destinations moving the token to both the *ChannelCBD* and the *ChannelStride*.

When the token arrives at the *ChannelStride* and *ChannelCBD*, we know that no other channel has currently access to the memory read setup interface. This guarantees that a situation cannot occur when the ReadAddress and the ReadCBDIn belong to two different channels. Hence the ChannelStride and

ChannelCBD submit their read request to the memory and wait for the request to be processed. After a read reuest is submitted a new token is produced allowing other channels to submit their requests.

In step 4 the read operation is completed and the accompanying cbd is sent to the fetch unit. It contains instruction 5 moving a token literal to the NextReadData terminal to pull the data to the Sorter input, and instruction 6 moving the requested data to the sorter input.

At the end of the data loading cycle, the channel is prepared for the next iteration. The code for this is in the *MOVE_DATA* code bag and it sets up the ChannelCBD and ChannelStride as shown in steps 1 and 2 in Figure 3.18.

3.4.2 Loading Next Chunk

When a channel has finished reading a chunk it will request a new chunk. A chunk is identified by the address of the first data in the chunk. Loading the next chunk means setting the ChannelStride to the offset at which the new chunk is starting, shown in Figure 3.22.

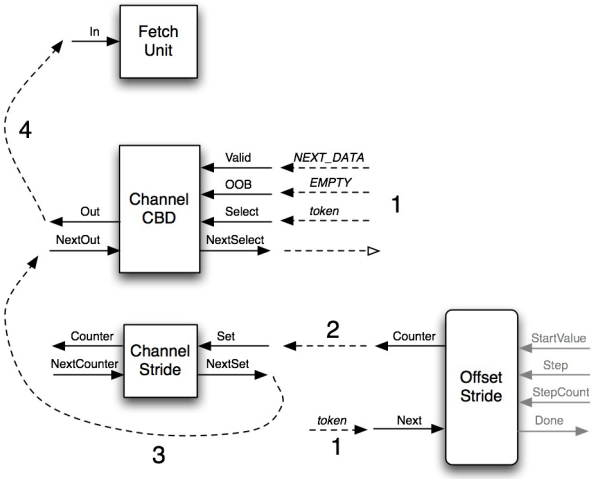


Figure 3.22: Channel loading next chunk

All the instructions shown in Figure 3.22 are contained in the *NEXT_CHUNK* code bag, as shown in Figure 3.23.

The chunk offsets are generated by the *OffsetStride*. It is an instance of the *Stride* ship.

Stride

The *Stride* ship generates a sequence of equally spaced integer numbers and is shown in Figure 3.24.

The behavior of the *Stride* ship is captured by the following concurrent program:

```
codebag NEXT_CHUNK_TO_CHANNEL_1 {
    // setup the OffsetStride
    move (token) -> OffsetStride.Next
    move OffsetStride.Counter -> Channel1Stride.Set

    // trash the NextSelect token
    move Channel1CBD.NextSelect -> bitBucket

    // setup the ChannelCBD
    move (token) -> Channel1CBD.Select
    move (NEXT_DATA_TO_CHANNEL_1) -> Channel1CBD.Valid
    move (ERROR) -> Channel1CBD.OOB

    // make sure ChannelCBD will produce the cbd
    // AFTER ShannelStride is set up
    move Channel1Stride.NextSet -> Channel1CBD.NextOut

    // load the next data
    move Channel1CBD.Out -> fetcher.In
}
```

Figure 3.23: *NEXT_CHUNK* code bag loading the next chunk to channel 1.

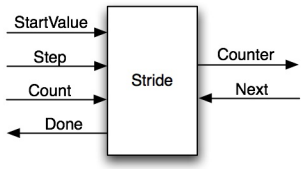


Figure 3.24: The Stride ship

$Stride =$
$(StartValue?start \parallel Step?step \parallel Count?count)$;
if $valid(start) \wedge valid(step) \wedge valid(count)$ **then**
  $counter, i := start, 0$;
  while $i < count$ **do**
    $Next?token$; $Counter!counter$; $counter, i := counter + step, i + 1$
  end while
  $Done!token_{valid}$
else
  $Done!token_{oob}$
end if
$Stride$

□

Coming back to Figure 3.22, whenever a channel requests the next chunk, it sends a token to the Next terminal of the OffsetStride. The OffsetStride then returns the offset address of the next chunk. Setting up the OffsetStride for a new run is described in detail in Section 3.5.1.

Thus in the first step the offset is loaded from the OffsetStride into the ChannelStride. Simultaneously the ChannelCBD is prepared to release the code

bag descriptor which will load the first data from the new chunk, following the steps described in Section 3.4.1.

It is important that the next data is requested only *after* the ChannelStride has been setup with the offset of the new chunk, otherwise poking the *ChannelStride* with a token will generate the address of the next data from the old chunk (i.e. an address outside of the chunk). Therefore we move the *NextSet* token from the ChannelStride to enable the output of the ChannelStride only after the new offset was received by the ChannelStride [6].

3.5 Sequence in Write Channel

The next run is loaded *after* all data from the previous run was written to the memory. Thus we need to bring *sequence* into the program execution.

The write channel is associated with the output of the last sorter ship. It writes data to the memory using the *MemoryWrite* ship. After all data has been written to memory, the channel checks how many chunks were written. In case of a single chunk, the sorting is complete (see Section 3.6), otherwise the next run is loaded (see Section 3.5.1).

MemoryWrite

The *MemoryWrite* ship is a memory controller allowing to write data to memory. It can pipeline several write requests and process them in any order. It is equipped with a pipeline interface and a cbd interface for submitting a code bag descriptor which will be published at the cbd output when the corresponding operation has completed.

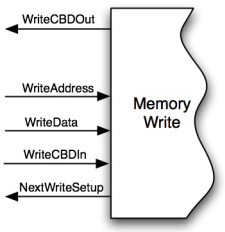


Figure 3.25: The MemoryWrite ship

The behavior of the *MemoryWrite* ship is captured by the following concurrent program:

[6] Note the passive use of the pipeline interface in this case, where we treat the token as an *acknowledgment* for receiving the last data, rather than a *request* for he next data.

This is a subtle, yet important difference, since at the end of an active pipeline cycle there is a token waiting on the input pipeline interface, while at the end of a passive pipeline cycle the token is used up. When programming Fleet it is important to keep track how the pipelines are used.

initially i = 0;

$MemoryWrite =$
$i := i + 1 \mod bufferSize$
$(WriteAddress?address[i] \parallel WriteData?data[i] \parallel WriteCBDIn?cbdIn[i])$;
if $valid(address[i]) \wedge address$ within memory range **then**
  $(NextWriteSetup!token_{valid}$; $MemoryWrite) \parallel$
  $WriteCBDOut!cbdIn[i]$
else
  $NextWriteSetup!token_{oob}$; $MemoryWrite$
end if

□

Note that $address[i]$, $data[i]$ and $cbdIn[i]$ each point to a location in a buffer with size set to *bufferSize*. The buffers are introduced to allow the *MemoryWrite* ship to pipeline several write requests. To simplify the program we assume that data can be read from an input to a buffer location i only after the previously buffered data at $cbdIn[i]$ was written to$WriteCBDOut$.

Just like the read channels, the write channel has a *WriteStride* and *WriteCBD* ships, equivalent to the ChannelStride and ChannelCBD ships. The WriteStride is an instance of the Stride ship and the WriteCBD is an instance of the *Constant* ship, which generates the *EMPTY* code bag descriptor whenever poked by a token.

The write channel writes data to the memory similarly to the read channels reading data from the memory. We will therefore skip the discussion of writing data and focus on loading the next chunk.

3.5.1 Loading Next Run

Identifying when to load the next run

A run is finished when all the data comprising the merged chunks is written back to the memory. A *WriteCounter* is introduced which keeps track of the number of elements written to the memory, by counting the code bag descriptors returned by the memory write interface. When the last code bag descriptor arrives at the WriteCounter it triggers the Rendezvous ship to release the *NEXT_RUN* code bag descriptor.

The *WriteCounter* is an instance of the *Counter* ship and the *Rendezvous* is an instance of the *Rendezvous* ship.

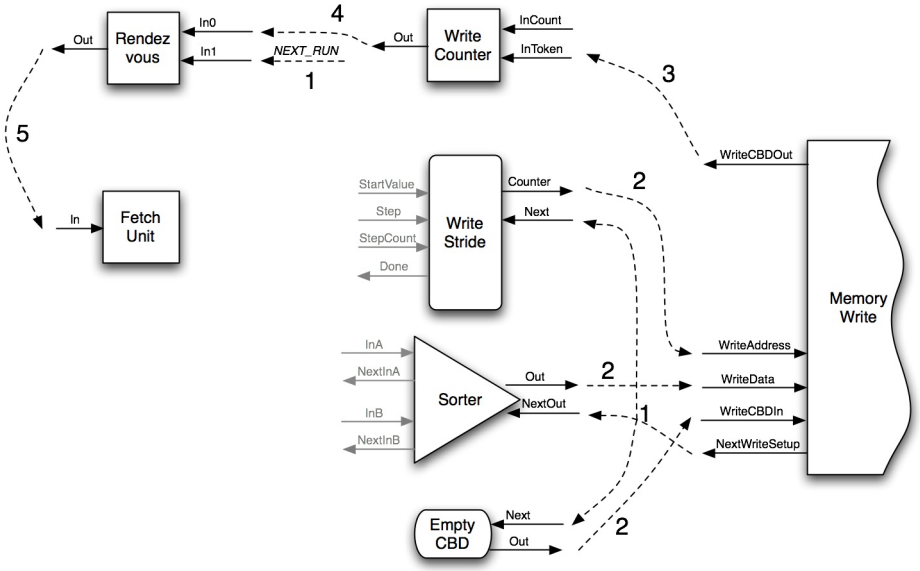


Figure 3.26: Loading next run

Counter

The *Counter* ship counts the incoming tokens and when a the number of tokens reaches a specified number it generates a token on its output.

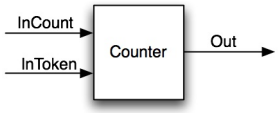


Figure 3.27: The Counter ship.

The behavior of the *Counter* ship is captured by the following concurrent program:

$Counter =$
$InCount?count$;
if $valid(count)$ **then**
  $counter := count$;
  while $0 < counter$ **do**
    $InToken?token$; $counter := counter - 1$
  end while
  $Out!token_{valid}$
else
  $Out!token_{oob}$
end if
$Counter$

Rendezvous

A *Rendezvous* ship allows to synchronize events.

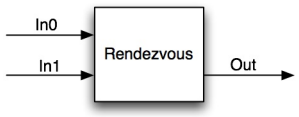


Figure 3.28: The Rendezvous ship.

The behavior of the *Rendezvous* ship is captured by the following concurrent program:

$$Rendezvous = (In0?data0 \parallel In1?data1)\ ;\ Out!data1\ ;\ Rendezvous$$

□

For every run the WriteCounter is initialized with the total number of elements which are going to be written. This number is N plus the number of OOB data terminating each chunk written to the memory.

Setting up the next run

This brings us to the calculations necessary to initialize the strides for each run. In our Fleet we store all together five variables:

- N is the number of elements to be sorted
- *ChunkCount* is the number of chunks in the current run
- *ChunkSize* is the number of elements in the read chunk in the current run
- *OffsetRead* is the memory address of the first data in the first read chunk of the current run
- *OffsetWrite* is the memory address of the first data to be written in the current run

These are stored in single stage FIFOs. To perform a calculation on the variable, the value is taken from the output of a fifo, modified by some ships and finally written back to the input of the same fifo. Storing a variable in single stage fifos guarantees that reading and writing values to the variable will alternate. If we were to introduce a register ship, which would allow the value to be overwritten, synchronization of arithmetic operations involving the variables (described below) would be more difficult.

Before each run the variables are used to initialize the OffsetStride on the reading side, and the WriteStride and WriteCounter on the writing side, as

shown below:

- setup OffsetStride:
  - StartValue := *ReadOffset*
  - Step := *ChunkSize* + 1
  - StepCount := *ChunkCount*
- setup WriteStride:
  - StartValue := *WriteOffset*
  - Step := 1
  - StepCount := N + *ChunkCount*/C
- setup WriteCounter:
  - Set := N + *ChunkCount*/C

In OffsetStride setup 1 added to the *ChunkSize* stands for the OOB data terminating each chunk. In WriteStride and WriteCounter setup C is the capacity of the sorter.

This brings us to the calculations needed to run our fleet. We have two requirements regarding the sequence of events:

- Before setting up the OffsetStride we need to increase the *ChunkSize* by factor C, since between the runs, while the number of chunks decreases, the chunk size grows.
- Before setting up the write channel, we need to decrease the chunk number. Since we have C read channels, each C chunks will be merged into 1 chunk. This will reduce the number of chunks by factor C, hence we need to divide *ChunkCount* by C.
- Before setting up the next run the *ReadOffset* and the *WriteOffset* need to be swapped.

The following sequence of tasks guarantees the requirements. The numbers define the sequence between groups of tasks. Tasks assigned to the same number may execute concurrently.

1. setup OffsetStride
2. *ChunkSize* := *ChunkSize* × C
   ChunkCount := *ChunkCount*/C
3. setup WriteStride
   setup WriteCounter
4. *ReadOffset*, *WriteOffset* := *WriteOffset*, *ReadOffset*

If C is a power of 2, the multiplication and division by factor C can be easily done by shifting the bits representing the numbers to the left and to the right.

The arithmetic operations in the above assignments require two additional ships: an Adder and a Shifter. The Adder is described in Section 2.1.1.

Shifter

The *Shifter* ship performs a bit shift on the input on *In*. The number of bits to be shifted and direction is given by *Shift*.

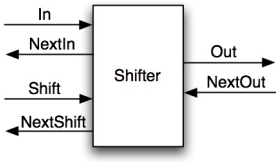


Figure 3.29: The shifter ship.

The behavior of the *Shifter* ship is captured by the following concurrent program:

$$
\begin{array}{l}
Shifter = \\
NextOut?token\ ; \\
(In?in \parallel Shift?shift)\ ; \\
\textbf{if}\ valid(in) \wedge valid(shift)\ \textbf{then} \\
\quad \textbf{if}\ 0 \leq shift\ \textbf{then} \\
\quad\quad Out!in \ll shift \parallel NextIn!token_{valid} \parallel NextShift!token_{valid} \\
\quad \textbf{else} \\
\quad\quad Out!in \gg -shift \parallel NextIn!token_{valid} \parallel NextShift!token_{valid} \\
\quad \textbf{end if} \\
\textbf{else} \\
\quad Out!token_{oob} \parallel NextIn!token_{oob} \parallel NextShift!token_{oob} \\
\textbf{end if} \\
Shifter
\end{array}
$$

□

An assignment

OffsetStride.Step := *ChunkSize* + 1

is realized in Fleet by the following instructions

move ChunkSize.Out → Adder.In0
move (1) → Adder.In1
move Adder.Out → OffsetStride.Step, ChunkSize.In

The other assignments follow a similar pattern.

3.6 Initializing and Finalizing the Fleet

Initializing the fleet consists of priming the pipeline interfaces and setting up standing moves. As we mostly use the active pipeline approach, we expect a token to be present at the input pipeline interface before we send a new input to the ship. Since initially the pipelines are passive, i.e. are waiting for new input to arrive before producing a token, we send dummy inputs and either reuse any outputs to prime other ships, or dump them to the bitBucket.

Finalizing deals with emptying the fleet of any residue control data and disassembling remaining standing moves. Sorting is complete when the last data from the last run was written to the memory. At this point the Fleet needs to be finalized, i.e. cleaned up. In case of our sorter Fleet we know which sources in the switch fabric contain data (e.g. pipeline tokens and values inside the fifos storing *ChunkSize*, *ReadOffset*, etc.), so the fleet can be emptied by releasing a code bag which will move all residue data to the bitBucket and dissasemble any standing moves by feeding *OOB* data through the corresponding sources.

3.7 The primitives

During the discussion in Sections 3.3 and 3.4 we introduced several *programming primitives* which helped us solving the encountered problems. A primitive is defined by a set of ships together with a set of move instructions performing the function of the primitive.

3.7.1 Code Bag Descriptor Interface

We use the code bag descriptor interface to avoid the conflicting sources problem, where two instructions sharing the same source

move $a \rightarrow x$
move $a \rightarrow y$

are executed concurrently.

The code bag descriptor interface allows to transfer control together with a data stream. A ship may be equipped with an input and output code bag descriptor interface, as show in Figure 3.30.

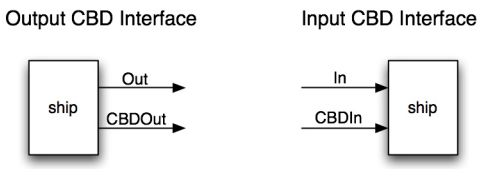


Figure 3.30: Code bag descriptor interface.

The output code bag descriptor interface produces a code bag descriptor accompanying every data on *Out*. The input code bag descriptor interface expects a code bag descriptor with every data on *In*. A ship will usualy have both input and output interfaces. A *CBDIn* or *CBDOut* can be assigned to several inputs or outputs.

3.7.2 Pipeline Interface

We use the pipeline interface to avoid the conflicting destinations problem, where two instructions sharing the same destination

move $x \rightarrow a$
move $y \rightarrow a$

are executed concurrently.

The pipeline interface communicates tokens and has two components: the input and the output interface, as shown in Figure 3.31.

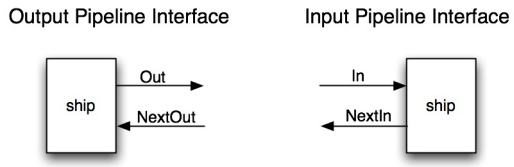


Figure 3.31: The pipeline interface

An output interface extended with the pipeline interface will not produce the output before it receives a token on the output pipeline interface. the output pipeline interface is agnostic about the validity of the incoming token and treats both valid and OOB tokens alike.

An input interface extended with the pipeline interface will announce that the ship is ready to receive the next input by producing a token on the input pipeline interface. If the incoming data is OOB, the input pipeline interface will produce an OOB token, otherwise a valid token will be produced.

Note how we exploit the fact that a move instruction cannot complete until the data at the source is available, to bring sequence into the program execution.

3.7.3 Rendezvous

We use the Rendezvous primitive to bring sequence to the execution of a concurrent Fleet program, as shown in Figure 3.32. It employs a Rendezvous ship and exploits the multiple destinations in a move instruction.

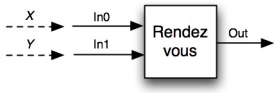


Figure 3.32: The Rendezvous primitive implementing X ; Y

The rendezvous primitive allows to execute instructions in sequence, by loading the next code bag after an instruction has executed. Assume we have a code bag

```
A {
    move a → x
    move b → x
}
```

and we want to execute $a \rightarrow x$ followed by $b \rightarrow x$. The recipe is as follows:

- Wrap the $b \rightarrow x$ move in a code bag with descriptor B.
- Set up the *rendezvous* ship with the B code bag descriptor, by adding the instruction $(B) \rightarrow rendezvous.In1$ to code bag A.

- Add instruction $rendezvous.Out \rightarrow fetch.In$ to code bag A, in order to fetch the code bag B whenever the *rendezvous* ship is triggered.
- Extend the $a \rightarrow x$ move with multiple destinations $a \rightarrow x, rendezvous.In0$ to trigger the release of code bag B.

Since the Rendezvous ship is set up with a code bag descriptor it can implement event handling, where the incoming token represents an event, and the code bag descriptor identifies the corresponding event handler. The OOBSelector ships can be seen as an extension to the Rendezvous ship, allowing to select between two event handlers depending on the validity of the event token.

3.7.4 Rendezvous with Counter

We use a combination of a Rendezvous ship and a Counter ship to release a code bag descriptor after counting certain number of events occurred.

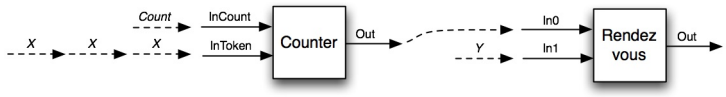


Figure 3.33: The Rendezvous with Counter primitive

It allows to implement in Fleet the following

for $i = 1$ **to** *Count* **do** X **end for** ; Y

The recipe is similar to the Rendezvous primitive. The difference lies in having each event X send a trigger to the Counter ship, and the Counter ship trigger the Rendezvous to release the next code bag.

A Rendezvous with Counter is an example of a generalization of the Rendezvous primitive. The Counter ship can be replaced by a different ship, for example one detecting an OOB in a data stream. Substituting different ships for the Counter allows to capture and handle different events.

3.8 Discussion

General ships We created a fleet using general purpose ships, such as adders, selectors and strides, without resorting to special purpose ships designed specifically for this particular sorting fleet.

Data Driven The order in which the chunks are assigned to channels does not matter, as long as all chunks in a run are assigned to one and only one channel. This allows the read channels to operate independently of each other and drive their own dataflow, synchronized only at the shared memory read interface and the offset stride generating the chunk offsets.

The *OOBSelector* and *Compare* ships allowed to design a data driven program, where the program execution is governed by the validity of the data in a data stream, or relative values in two data streams.

Synchronization Memory is a shared resource and access to it needs to be synchronized. We used the pipeline and code bag descriptor interfaces to do the job, relying on the features of the underlying Fleet architecture, such as asynchrony in the switch fabric.

Sequence We introduced sequential computation in several ways:

- The stride ship is an example of a functional unit which implements a sequential computation. Other more complicated ships can be imagined.
- The pipeline interface allows to arrange ships in pipelines, where computation is progressing through the ships according to the pipeline.
- We used a combination of *Rendezvous* and *Counter* ships for loading the next run after all data was written to the memory. The *Rendezvous* allows to dispatch a particular computation upon an event, while the *Counter* is responsible for identifying the event by counting the number of data items written to the memory. Note that the Counter ship can be substituted by a different ship identifying a different event, for example a ship which would accept a stream of data and generate a token whenever it encounters an OOB data.

Congestion Control Congestion in the switch fabric was indirectly tackled by the pipeline and code bag descriptor interfaces. The pipeline interface prevents sources from flooding the destinations with data, and thus prevents congestion in the destination horn. The code bag descriptor interface allows to moderate fetching of instructions and thus prevents congestion in the instruction horn.

Limitations Our sorter Fleet can sort sequences with length N, where N is a power of C. In case of three sorting units as shown in Figure 3.3, N needs to be a power of 4.

Despite the serious restriction on N, the sorter proves to be a good example for exposing different features and challenges of programming Fleet.

Chapter 4

Analysis

In this chapter we present a performance analysis aiming at revealing the effect of the primitives described in Chapter 3 and concurrency of the Fleet architecture on the performance of our sorting Fleet, and how it relates to hypothetical implementation on a traditional von Neumann machine.

We would like to gain insight into the behavior of our sorter Fleet depending on the parameters, such as the latency of the switch fabric, memory and other ships. For example, in case of a slow memory we expect the sorter to behave sequentially, offering little gain from concurrent computation. Depending on different parameter regimes we expect to find different bottlenecks and to identify in which situations certain ships and configurations of ships, such as the primitives described in the previous chapter, perform well.

4.1 Measuring Performance

We will look into the *runtime* performance measure of the Fleet sorter. When talking about runtime complexity it is important to distinguish between different performance metrics, depending on their use.

Problem complexity

The problem complexity sets a lower bound on the complexity for any algorithm solving the problem, assuming some computation model. It refers to the number of basic computation steps needed to solve the problem, disregarding any particular domain. The general binary-comparison-based sorting problem has complexity $O(N \log N)$.

Algorithm complexity

An algorithm represents a particular way of solving a problem. It assumes the same computation model as described by the problem analysis. The algorithm complexity sets a lower bound on the complexity of any implementation of the algorithm. In case of parallel sorting, the algorithm complexity refers to the time between reading the first element of the input unsorted sequence form the memory and writing the last element form the sorted sequence to the memory.

Assuming data in the memory is accessed sequentially, the lower bound on the runtime of merge sort is $O(N \log N)$.

Implementation complexity

The implementation complexity measures the performance of a particular mapping of a particular algorithm onto a particular machine. The computation model is more detailed than the model used by the algorithm. The implementation complexity can be modeled theoretically, or it can be measured empirically. Unlike the algorithm performance model hiding constant factor parameters in the O notation, the implementation model needs to take into account the details and costs incurred by mapping the program onto a machine, including the constant factors.

The theoretical model allows to predict the performance for different parameters, such as the problem size or the machine complexity. The empirical model measures the performance for several particular instances of the parameters, and checks whether the theoretical model is sound. Conversely, given a set of empirical results a theoretical model can provide us with an understanding of how these results came about.

4.1.1 Runtime

The performance of a concurrent implementation can be measured by the run time $T(P, N)$, which measures the time for solving a problem instance of size N on P processors. The run time of each processor i can be divided into

- $b_i(P, N)$: time busy computing
- $c_i(P, N)$: time spent on communication
- $y_i(P, N)$: idle time

Different processors may divide their time differently between the three tasks, but for all processors the total run time is the same. Hence for each processor i

$$b_i(P, N) + c_i(P, N) + y_i(P, N) = T(P, N) \tag{4.1}$$

4.1.2 Speedup

Performance of introduced concurrency can be approximated by the *speedup*. The speedup represents the improvement in speed due to concurrency and is given by

$$S(P, N) = \frac{T(1, N)}{T(P, N)} \tag{4.2}$$

The speedup on its own is not a good performance measure, as in some situations we could require a tremendous number of processors to get a marginal speedup. Thus to find the efficiency, the speedup should be scaled against the number of processors P. Of course it cannot grow indefinitely, as the problem size N poses an limit on the number of processors which can be effectively utilized.

4.1.3 Efficiency

Efficiency scales the speedup with the number of processors, and measures the quality of speed up. It is expressed as

$$E(P,N) = \frac{S(P,N)}{P}$$

- $S(P,N) < P$ is a *sub-linear speedup*, $E(P,N) < 1$
- $S(P,N) = P$ is a *linear speedup*, $E(P,N) = 1$
- $S(P,N) > P$ means a *super-linear speedup*, $E(P,N) > 1$

We will first derive a theoretical model of our sorter Fleet, compare it with the simulation results and then look what we can learn from this experiment about designing Fleet.

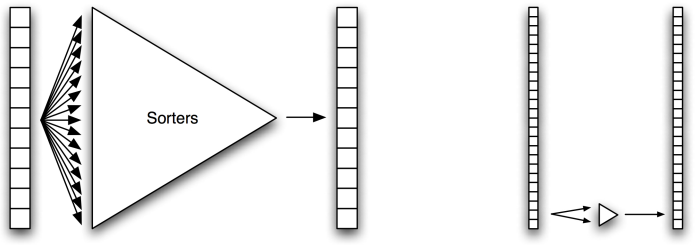
4.1.4 Some expectations

Before we embark on deriving the performance model, let us state our expectations for the performance model.

- We expect a speedup to increase with increasing P. If we were to sort a sequence of length N on $P = N$ processors, the merge sorting network would require a single run through the sequence taking N steps, thus giving a speed up of

$$\frac{T(1,N)}{T(N,N)} \approx \frac{N \log N}{N} = \log N$$

  With $\frac{N}{P}$ approaching 1 (Figure 4.1.a), we expect the speedup to approach $\log N$. On the other hand, with $\frac{N}{P}$ approaching ∞ (Figure 4.1.b), we expect the speedup to approach 1. For $1 < P$ we expect a speedup to be greater than 1, as otherwise it would make little sense to increase the number of processors.



(a) (b)

Figure 4.1: Different ratios between P (size of the hardware) and N (size of the input sequence), (a) showing $\frac{N}{P}$ approaching 1 and (b) showing $\frac{N}{P}$ approaching ∞.

- We also expect the relative speeds of the memory and the processors to be reflected in the performance model. With a slow memory there will be little gain from a large network of concurrent sorter ships, as the data flow will be serialized at the memory bottleneck.

4.1.5 The Complexity Measure

In the discussion above the complexity of the system was represented by the number of processors P. How does P translate to our sorter Fleet?

We increase the complexity of our sorter Fleet by adding sorters and arranging them in a binary tree, as shown in Figure 4.2. Let S be the number of sorter ships in our Fleet. We could let P represent the number of sorters. However, while increasing the number of sorters the number of channels also increases and thus also the number of ships implementing the channels[1]. Should we count every ship we add? This would not be fair as different ships have different complexities.

It is important that the relationship between the complexity measure and the real complexity of the machine is linear. We will measure the complexity of a sorter Fleet by the number of read channels C. Each Fleet contains 2 additional memory interface ships, MemoryRead and MemoryWrite. Further, if we arrange the sorters in a binary tree, then $S = C - 1$. The total number of ships in a sorter Fleet with C channels is thus

$$C + (C - 1) + 2 = 2C + 1$$

If we assume that each ship translates to a fixed number of transistors, our complexity measure C is linearly related to the real size complexity of our machine.

4.1.6 The Run Time Measure

The runtime measure $T(P, N)$ from the discussion above becomes $T(C, N)$, and Equation 4.1 becomes

$$T(C, N) = b_i(C, N) + c_i(C, N) + y_i(C, N) \tag{4.3}$$

The reference measure for speedup and efficiency becomes $T(2, N)$, measuring the runtime of a a Fleet containing a single sorter and *two* channels. Remember that efficiency attempts to relate the *factor* by which the speedup increases to the *factor* by which the hardware complexity increases. For example, a super linear speedup is attained when the speedup increases by a factor greater than 2, with the hardware complexity increasing by a factor of at most 2.

As the reference measure for speedup is $T(2, N)$, the equation for efficiency becomes

$$E(C, N) = \frac{S(C, N)}{\frac{C}{2}}$$

[1]Note that every read channel is implemented by a Stride and an OOBSelector ship.

4.2 Empirical performance model of the sorter Fleet

Let us start with the performance model based on the simulation results of our Sorter Fleet.

4.2.1 Archsim Simulator

We simulated our sorter Fleet in the *ArchSim* simulator [3, 4, 6, 7, 29]. ArchSim is a tool being developed at Sun Microsystems aiming at simulating Fleet. It provides:

- a library of predefined ships,
- a library of predefined switch fabrics (currently only the horn-funnel switch fabric),
- a Java API for implementing new ships,
- an XML interface for specifying Fleet configurations. A configuration is defined by a set of instances of components (such as ships and the switch fabric) and the connections between them. The components can be customized by specifying parameters, e.g. the size of the horn and funnels in the switch fabric or the delays in ships between consuming inputs and producing outputs.
- A compiler which compiles a Fleet program into Archsim binaries. These can then be executed on the Fleet specified in the configuration file containing the ships.

4.2.2 Simulation Results

In this section we present the simulation results aiming at measuring the increase in speedup of our sorter Fleet when increasing its complexity.

The simulation results are shown in Table 4.1. Note that for certain configurations of N and C no results are given. This is due to the limitation of our sorter Fleet, which can only sort sequences of size which is a power of C, i.e. where $N = C^x$ for $x \in \mathbb{N}$.[2]

4.2.3 Discussion

Tables 4.2 and 4.3 show how the speedup $S(C, N)$ and efficiency $E(C, N)$ vary with C and N.

We can identify two trends in Table 4.3.

1. With increasing C the efficiency first increases and then decreases.
2. With increasing N the efficiency first increases and then decreases.

[2]While this is a strong restriction, the current design acts as a good example for several aspects of Fleet design. The goal of this thesis was not to implement an efficient Fleet sorter, but to use sorting as an example problem in the investigation into programming Fleet.

N	$T(2,N)$	$T(4,N)$	$T(8,N)$	$T(16,N)$
4	69690	37650	-	-
8	150850	-	56218	-
16	343108	131846	-	73706
64	1806168	652052	384280	-
256	9309178	3549654	-	1400320
512	20816080	-	4626066	-
1024	46112020	18566280	-	-

Table 4.1: Simulation run times $T(C,P)$ measured in simulation time units for different values of N and C

N	$S(4,N)$	$S(8,N)$	$S(16,N)$
4	1.9	-	-
8	-	2.7	-
16	2.6	-	4.7
64	2.8	4.7	-
256	2.6	-	6.6
512	-	4.5	-
1024	2.5	-	-

Table 4.2: Speedup $S(C,N)$ for different values of N and C

Both trends correspond to the expectations from Section 4.1.4, which suggests that the simulation results are correct. To gain further confidence in the results we will try to derive a theoretical performance and compare with the empirical findings.

4.3 Theoretical performance model of the sorter Fleet

As discussed in Section 4.1.6 the runtime of our sorter Fleet is given by

$$T(C,N) = b_i(C,N) + c_i(C,N) + y_i(C,N)$$

We will try to express $T(C,N)$ in terms of the machine complexity C and problem size N.

4.3.1 $\mathbf{b_i}$(C,N)

The term $b_i(C,N)$ in Equation 4.2 represents the time spent by processor i on computation. Processor refers to a ship or collection of ships represented by a circle in Figure 4.3. It can be expressed as a product of the number of computation steps $n_{b_i}(C,N)$ and the time needed to perform each computation step t_i. The time taken by a processor i to perform a single step does not depend on C or N. Hence

$$b_i(C,N) = n_{b_i}(C,N) \cdot t_i$$

N	$E(4,N)$	$E(8,N)$	$E(16,N)$
4	0.9	-	-
8	-	0.7	-
16	1.3	-	0.6
64	1.4	1.2	-
256	1.3	-	0.8
512	-	1.1	-
1024	1.2	-	-

Table 4.3: Efficiency $E(C,N)$ for different values of N and C

Each processor performs a single operation step per input data (e.g. the sorting ships perform a single merge step and read channels perform a single memory-to-channel read step per data item passing through). Hence the number operations needed to sort the input sequence is equal to the total number of data elements flowing through each processor in the Fleet. The time complexity of an operation differs between the processors and is indicated by the different t values in Figure **??**. There are two types of data flowing through the sorter Fleet: the valid data and the OOB data representing end of chunk. Hence

$$n_{b_i}(C,N) = Data_i(C,N) + OBB_i(C,N) \tag{4.4}$$

4.3.2 c_i(C,N)

Similarly, the term $c_i(C,N)$ in Equation 4.2 representing the time spent by processor i on communication, can be expressed as a product of the number of communicated data elements $n_{c_i}(C,N)$ and the time needed to communicate each data t_c. The time it takes to communicate a single element includes the time to perform the move instructions and fetch code bag descriptors[3]. It is determined by the switch fabric and is independent of i, C and N. Hence

$$c_i(C,N) = n_{c_i}(C,N) \cdot t_c$$

Since every data element flowing through a ship is communicated exactly once, we have $n_{b_i}(C,N) = n_{c_i}(C,N) = n_i(C,N)$. Putting it together into Equation 4.3 we get

$$T(C,N) = n_{b_i}(C,N)\cdot t_i + n_{c_i}(C,N)\cdot t_c + y_i(C,N) = n_i(C,N)(t_i+t_c) + y_i(C,N)$$

According to Equation 4.1, $T(C,N)$ is the same for all processors, therefore different processors differ only in the distribution of runtime between $b_i(C,N) + c_i(C,N)$ and $y_i(C,N)$. Since $b_i(C,N) + c_i(C,N)$ is the fixed part and $y_i(C,N)$ the variable part, the processor with the largest $b_i(C,N)+c_i(C,N) = n_i(C,N)(t_i+t_c)$ will determine $T(C,N)$. As t_i and t_c are fixed, we will look at the $n_i(P,N)$ term.

[3]Congestion and arbitration in the switch fabric is not taken into account

4.3.3 $\mathbf{n_i(C,N)}$

The term $n_i(C, N)$ defines the total work done by processor i. With increasing P we expect less work to be done by each processor i, as otherwise it would make little sense to increase the number of processors.

As we mentioned in Equation 4.4,

$$n_i(C, N) = Data_i(C, N) + OBB_i(C, N)$$

Let us look closer at the terms $Data_i(C, N)$ and $OOB_i(C, N)$

$\mathbf{Data_i(C,N)}$

Let R be the number of runs needed to completely sort the input sequence, and let $data_i$ be the number of valid data items passed through a processor i in a single run. Then

$$Data_i(C, N) = data_i(C, N) \cdot R \tag{4.5}$$

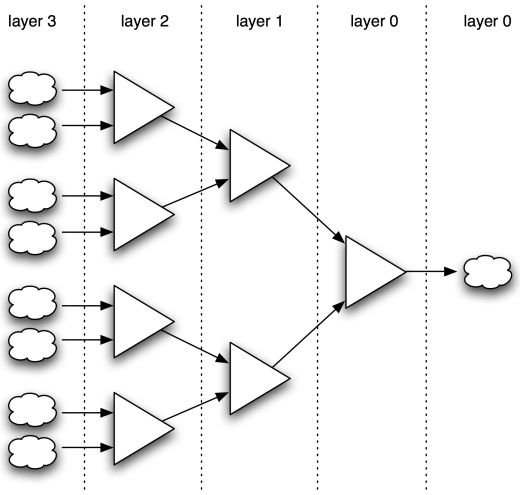


Figure 4.2: Sorters arranged in a binary tree

As mentioned before, a Fleet with P sorters arranged in a binary tree as shown in Figure 4.2, has $C = P + 1$ read channels. In each run the chunks increase in length by a factor C, while the number of chunks decreases by factor C. The sorting process starts with N chunks and goes on until there is a single chunk left, hence the number of runs is given by

$$R = \log_C N \tag{4.6}$$

For a given C there are $\log_2 C$ layers in the binary tree of sorters (as shown in Figure 4.2), with the root node being at level 0 and the leaves at level $\log_2 C - 1$.

Since in each run N elements enter and leave the sorter network, there are N elements passing through each layer. Each layer contains a different number of sorters and thus the elements are distributed differently between the sorters. In layer l_i there are 2^{l_i} processors with each handling $\frac{N}{2^{l_i}}$ elements.

Coming back to the number of valid data items passed through processor i in a single run, it is given by

$$data_i(C, N) = \frac{N}{2^{l_i}} \tag{4.7}$$

Combining Equations 4.5 - 4.7 we get

$$Data_i(C, N) = data_i(C, N) \cdot R = \frac{N}{2^{l_i}} \cdot \log_C N$$

$\mathbf{OOB}_i$(P,N)

Let us see how many OOB data are processed in total by the sorter Fleet. Every layer decreases the number of tokens by factor 2. If there were $C = N$ channels, then the number of OOB data processed by the sorter Fleet would equal

$$\sum_{i=0}^{\log_2 N} 2^i = 2N - 1$$

In the case when $C < N$, data cannot be stored inside the network and needs to be written to the memory between the runs. In each run all data, including the OOB sentinels, is read from and written to the memory. For larger C less runs are needed to completely sort a sequence, hence the total number of OOB data handled by the Fleet decreases for increasing C. In a given run, the number of OOB data read from memory equals to the number of OOB data written to the memory in the previous run, with the first run reading N and the last run writing 1 OOB data.

Table 4.4 shows the number of OOB data handled by all ships in a given layer, for $N = 256$ and varying C.

Layer	$C = 2$	$C = 4$	$C = 16$
0	$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$	$64 + 16 + 4 + 1$	$16 + 1$
1	$256 + 128 + 64 + 32 + 16 + 8 + 4 + 2$	$128 + 32 + 8 + 2$	$32 + 2$
2	-	$256 + 64 + 16 + 4$	$64 + 4$
3	-	-	$128 + 8$
4	-	-	$256 + 16$

Table 4.4: Number of OOB data handled by a layer for $N = 256$ and varying C

From the table we can see that the number of OOB items handled by layer l is

$$\sum_{i=1}^{\log_C N} \frac{C^i}{2^{\log_2 C - l}} = \frac{2^l(N-1)}{C-1}$$

However, for larger C, the greater total number of OOB handled by the Fleet is distributed between a larger number of processors. As there are 2^{l_i} processors in layer l_i, the number of OOB data handled by processor i is given by

$$OOB_i(C, N) = \frac{2^{l_i}(N-1)}{2^{l_i}(C-1)} = \frac{N-1}{C-1}$$

Note that every processor handles the same number of OOB elements, depending only on C and N and not on i.

4.3.4 y_i(C,N)

The runtime $T(C, N)$ is determined by the busiest processor, i.e. processor i with the smallest $y_i(C, N)$ term. Depending on the t_i parameters some ship will form a bottleneck. It will be the busiest ship, as other ships will have to wait for the bottleneck ship (otherwise one of the busier ships would be the bottleneck).

If we assume that the slowest (and busiest) processor is continuously occupied between the time it receives the first element in a run and the time it communicates the last element in the run, then its idle time is equal to the idle time during initialization t_{init} plus the idle time during each run. In each run the processor is idle for t_r time before it receives is first data and for t_s time after it sends its last data. If t_r and t_s measure the time it takes a single element to travel from the memory to the processor and from the processor to the memory, then their sum is equal to the time needed to propagate a single data element through the sorter t_{prop}. The idle time during initialization t_{init} is fixed. The propagation time through the sorter Fleet depends on the depth of the sorter network, and thus is given by $t_c \log_2 C$. Hence

$$y_{busiest}(C, N) = t_{init} + t_{prop} \cdot R = t_{init} + t_c \log_2 C \cdot \log_C N = t_{init} + t_c \log_2 N$$

4.3.5 T(C,N)

As mentioned before, $T(C, N)$ is determined by the busiest processor with the largest $n_i(C, N)(t_i + t_c)$ term. The discussion above provided us with the ingredients to compute n_i, which is given by

$$n_i(C, N) = Data_i(C, N) + OBB_i(C, N) = \frac{N \log_C N}{2^{l_i}} + \frac{N - 1}{C - 1}$$

where l_i is the layer in which the processor i is residing.

Let us take a quick look whether the formula meets our expectations. It describes the total amount of work done by processor i. It has two components, the first one describing the amount of work due to data, and the second describing the amount of work due to the OOB sentinels. With increasing C both terms will decrease, so our expectation of less work per processor with increasing hardware complexity is met. In the extreme cases, when $C = N$ each processors will process 1 OOB sentinel and between 1 and N data elements (depending on the layer in which the processor resides).

As the busiest processor has no idle time beyond the initialization time $y_{busiest}(C, N)$, we can compute $T(C, P)$ by taking the maximum over all processors

$$T(C, N) = \max_i (n_i(C, N)(t_i + t_c) + y_{busiest}(C, N)) \tag{4.8}$$

Figure 4.3 shows the service times, the t_i's of the processors in a sorting Fleet with 4 read channels.

We can expand Equation 4.8 by substituting for i the components from Figure 4.3.

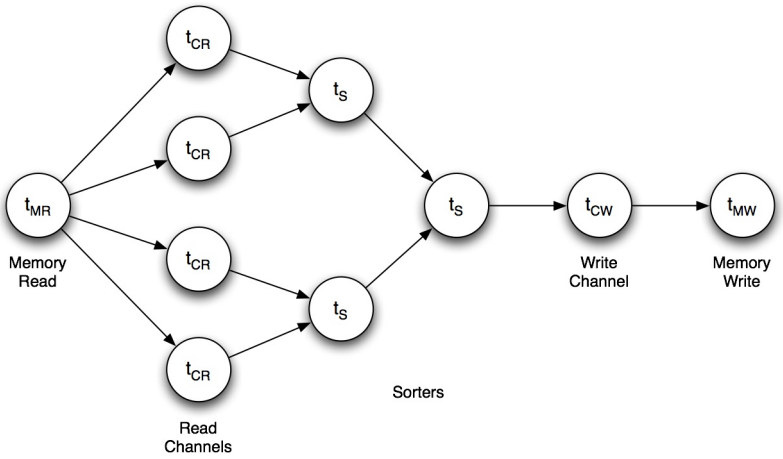


Figure 4.3: Service times in a sorter Fleet with 4 read channels.

$$
\begin{aligned}
T(C,N) &= \max_i(n_i(C,N)(t_i+t_c)+t_{init}+t_c\log_2 N) \\
&= t_{init}+t_c\log_2 N+\max_i(n_i(C,N)(t_i+t_c)) \\
&= t_{init}+t_c\log_2 N+\max(\\
&\quad n_{MR}(C,N)(t_{MR}+t_c), \\
&\quad n_{CR}(C,N)(t_{CR}+t_c), \\
&\quad n_S(C,N)(t_S+t_c), \\
&\quad n_{CW}(C,N)(t_{CW}+t_c), \\
&\quad n_{MV}(C,N)(t_{MV}+t_c)) \\
&= t_{init}+t_c\log_2 N+\max(\\
&\quad (\tfrac{N\log_C N}{1}+\tfrac{N-1}{C-1})(t_{MR}+t_c), \\
&\quad (\tfrac{N\log_C N}{C}+\tfrac{N-1}{C-1})(t_{CR}+t_c), \\
&\quad (\tfrac{N\log_C N}{1}+\tfrac{N-1}{C-1})(t_S+t_c), \\
&\quad (\tfrac{N\log_C N}{1}+\tfrac{N-1}{C-1})(t_{CW}+t_c), \\
&\quad (\tfrac{N\log_C N}{1}+\tfrac{N-1}{C-1})(t_{MW}+t_c)) \\
&= t_{init}+t_c\log_2 N+\max(\\
&\quad (\tfrac{N\log_C N}{1}+\tfrac{N-1}{C-1})(max(t_{MR},t_S,t_{CW},t_{MW})+t_c), \\
&\quad (\tfrac{N\log_C N}{C}+\tfrac{N-1}{C-1})(t_{CR}+t_c))
\end{aligned}
\tag{4.9}
$$

Note that there is only one term for all read channels. This reflects that the read channels operate concurrently. Also note how the sorter ship closest to the write channel determines the contribution of all sorter ships to the maximum runtime, since the other sorter ships in higher layers handle smaller data sets.

Let us see how the sorter Fleet will perform for different parameter regimes, i.e. for different values of the t parameters. We will denote with $\uparrow$ and $\downarrow$ the relative values of the parameters and chose t_{MW} as a representative for t_{MR}, t_S, t_{CW} or t_{MW}.

$t_{MW}\uparrow, t_{CR}\downarrow, t_c\downarrow, t_{init}\downarrow$: If the bottleneck is at the memory, one could think that deep networks are advisable, to do more of the computation internally

in the switch fabric, rather than via the memory. However, with the memory being the bottleneck, the sorter Fleet will behave *sequentially* and little is to be gained from introducing concurrent read channels. There will be a slight increase in efficiency for small C, since for increasing C more computation will take place in the network, rather than going through the memory (indicated in Equation 4.9 by fewer runs and fewer OOB sentinels). However, for large C the dataflow through the Fleet will be regulated by the slow memory interface.

$t_{MW} \downarrow, t_{CR} \uparrow, t_c \downarrow, t_{init} \downarrow$: If the bottleneck is at the read channels, then adding more *concurrent* channels may leverage the load from the existing channels and make better use of the other ships by feeding them data at a bigger rate. Hence with increasing C the efficiency should increase, until it reaches a point where the other ships cannot process the data provided by the read channels fast enough and the bottleneck shifts towards other ships.

$t_{MW} \downarrow, t_{CR} \downarrow, t_c \uparrow, t_{init} \downarrow$: If the bottleneck is at the switch fabric, it will increase the in-network computation time. In this regime it is more profitable to communicate data through the memory, rather than the switch fabric. Therefore increasing the number of read channels will lower efficiency.

$t_{MW} \downarrow, t_{CR} \downarrow, t_c \downarrow, t_{init} \uparrow$: If the initialization takes relatively long, then the sorter behaves sequentially and little can be gained from concurrency, which is exploited after the initialization.

The regimes presented above discuss only 4 examples of the 16 possible combinations. They represent the extreme cases with different bottlenecks. Other regimes share the characteristics of some of these regimes, depending on the relative magnitudes of the parameters.

Figure 4.4 shows a plot of $E(C, N)$ for different C and N, for $t_{MR} = 50$, $t_{CR} = 4t_c$, $t_S = 20$, $t_{CW} = 2t_c$, $t_{MW} = 50$, $t_c = 560$ and $t_{init} = 7400$. We selected the values to roughly correspond to the simulation parameters in the next section. Figure 4.5 shows two slices through the Figure 4.4.

At first glance the Figure 4.5.a looks suspicious, as for small C the efficiency increases beyond 100%, which suggests that with increasing C the amount of work which needs to be done decreases. This coincides with the discussion above, which states that for larger C less OOB sentinels need to be processed. For small C, the drop in the number of OOB sentinels is relatively large and we get a super linear speedup. However, when C increases further the difference becomes relatively small. At the same time the bottleneck shifts towards the memory making the sorter Fleet behave sequentially. Increasing the number of read channels even further will not speed up the sorting process and thus the efficiency drops.

The hump in Figure 4.5.b also looks suspicious. We would expect efficiency to be a more logarithmically shaped function of N, approaching a certain maximum. The initial increase in efficiency is due to the initialization overhead. Its impact on the runtime cannot be ignored for small N, but becomes negligible for large N. Then the hump occurs and the efficiency start to decrease with increasing N. It can be explained with the large t_{CW} parameter. A bottleneck at the

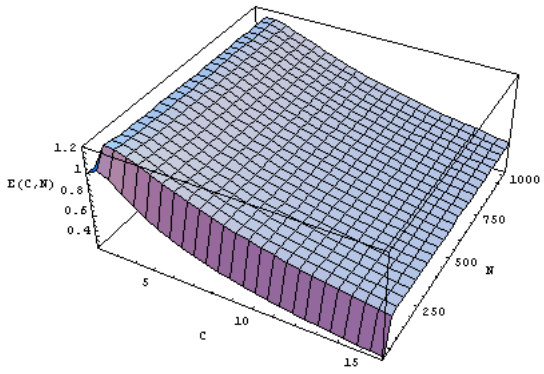
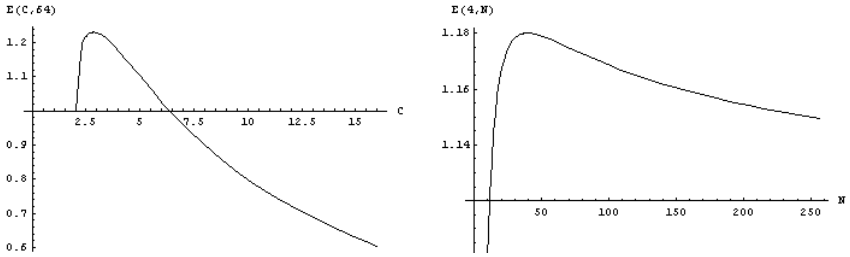


Figure 4.4: Anticipated $E(C, N)$ for $t_{MR} = 50$, $t_{CR} = 4t_c$, $t_S = 20$, $t_{CW} = 2t_c$, $t_{MW} = 50$, $t_c = 560$ and $t_{init} = 7400$



(a) (b)

Figure 4.5: Slices through Figure 4.4, (a) showing $E(C, 64)$ and (b) showing $E(4, N)$.

WriteChannel results in a sequential behavior of the whole Fleet sorter. Since the amount of work needed to sort a sequence of length N is $O(N \log N)$, the amount of work increases faster than N. As the Fleet is operating sequentially, the runtime increases faster for larger N, resulting in a drop in efficiency.

4.4 Discussion

In general, the theoretical performance model matches the empirical simulation results, which is very pleasing. Furthermore, both meet the expectations from Section 4.1.4. The exact numbers in the simulation results differ from the analytical results. It can be attributed to the additional details, which we did not take into account in our theoretical model, e.g. fetching of code bags or the influence of congestion in the switch fabric on the throughput.

4.4.1 Better than von Neumann?

An interesting question is how the Fleet implementation of the merge sort compares to an implementation on a von Neumann machine.

Our theoretical model takes into account the OOB sentinels used to control the dataflow through the sorter, which on a traditional von Neumann computer would be done with the program counter. Therefore our analysis is Fleet specific.

However, when we look at Equation 4.9 we recognize familiar terms also found in the von Neumann architecture. As the discussion of the different regimes points out, when the number of processors increases the bottleneck shifts towards the memory. Therefore the von Neumann memory bottleneck still remains in Fleet.

The advantage of Fleet lies in the ease with which a particular Fleet can be extended with additional memories, to alleviate the memory bottleneck. Since memory is treated like any other ship, the change is limited to a relatively simple modification of the Fleet program.

The question remains whether the merge sort problem was appropriate for Fleet and whether the Fleet sorter presented in this thesis fully exploits the strength of the Fleet architecture. It could very well happen that Fleet outperform the von Neumann architecture significantly in other problems. The future experiments will show.

4.4.2 Influence of the primitives

In the previous chapter we have introduced primitives to enhance programability of Fleet, but they come at a certain price. All three primitives can be found in the read channels and increase the time needed to read data from the memory by at least 5 passes through the switch fabric.

When introducing these primitives we trade off performance for correctness of execution, which can be seen by the large values of t_{CR}. Synchronization of data flows requires pipeline interfaces which double the latency. In case of a Fleet where the read channels are the bottleneck, this increase in latency can be compensated by adding more read channels and thus overlapping the execution of concurrent channels. At some point, however, the limit is reached where adding more channels does not pay off, as the bottleneck shifts to a different part of the Fleet.

Chapter 5

Conclusion

In this thesis we researched the Fleet architecture, focussing on the programability aspect. We selected a particular problem of merge sorting and presented a set of primitives which proved to be useful for designing and programming a sorter Fleet. Each primitive consisted of a set of ships and a template for move instructions required to operate the ships. The primitives allowed to retain concurrency in the execution of a program on the Fleet architecture, which provides no mechanism to compose atomic operations other than the single move instructions.

We evaluated the performance of the primitives by simulating the sorter Fleet in the ArchSim simulator. To gain understanding and support for the empirical results, we derived a theoretical performance model for the sorter Fleet. We were excited to see that the theoretical results confirmed the simulation results. In the analysis we have focused on the data flow, without going into the details of for example instruction fetching or congestion in the switch fabric.

Regarding Fleet in general, we hoped to find that Fleet can outperform the traditional von Neumann architecture. It tackles the program counter bottleneck by executing instructions as soon as their operands become available. In order to optimize the communication between the ships, it provides a single move instruction, providing the programmer with explicit control over the communcanition. Unfortunately, our results indicate that Fleet is still haunted by the memory bottleneck. Since Fleet regards the memory just as any other ship, however, the bottleneck can be remedied easier than on a von Neumann machine.

An interesting question for future research is whether these primitives are general enough to be reused in other Fleet designs. We have tried to abstract the essence of the encountered problems, reducing them to the lack of atomicity, manifested by the conflicting sources and destinations. Whether this applies to other designs can only be shown by designing other Fleets. An interesting application for the primitives may be a compiler from a higher level programming language, e.g. Lisp, to the Fleet single-instruction set.

Another interesting question for future research is how the Fleet architecture compares to other sequential and parallel computer architectures. At this stage little is known about its behavior in different situations. More detailed analysis is required, taking into account not only the data flow, but also the details of the switch fabric, instruction fetching and other.

Bibliography

[1] K. Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, 1990.

[2] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[3] I. Benko. ArchSim structure. Technical Report UCIB 2005-ib01, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, September 2005.

[4] I. Benko. GasP model in ArchSim. Technical Report UCIB 2005-ib02, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, September 2005.

[5] I. Benko. Introducing sequential behavior in Fleet. Technical Report UCIB 2005-ib05, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, September 2005.

[6] I. Benko. XML front end for ArchSim. Technical Report UCIB 2005-ib03, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, September 2005.

[7] I. Benko. FLEET assembly. Technical Report UCIB 2005-ib05, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, March 2006.

[8] I. Benko and J. C. Ebergen. Composing snippets. In *Concurrency and Hardware Design, Advances in Petri Nets*, pages 1–33, London, UK, 2002. Springer-Verlag.

[9] D. Bitton, D. J. DeWitt, D. K. Hsaio, and J. Menon. A taxonomy of parallel sorting. *ACM Comput. Surv.*, 16(3):287–318, 1984.

[10] W. S. Coates, J. K. Lexau, I. W. Jones, S. M. Fairbanks, and I. E. Sutherland. FLEETzero: An asynchronous switching experiment. *async*, 00:173, 2001.

[11] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[12] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *ISCA '75: Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, New York, NY, USA, 1975. ACM Press.

[13] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[14] J. Ebergen. Squaring the FIFO in GasP. In *ASYNC '01: Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems*, page 194, Washington, DC, USA, 2001. IEEE Computer Society.

[15] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, 1985.

[16] J. R. Gurd and D. F. Snelling. Manchester data-flow: a progress report. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 216–225, New York, NY, USA, 1992. ACM Press.

[17] R. A. Iannucci. Toward a dataflow von Neumann hybrid architecture. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 131–140, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[18] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[19] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[20] A. Peeters and K. van Berkel. Single-rail handshake circuits. In *ASYNC '95: Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, page 53, Washington, DC, USA, 1995. IEEE Computer Society.

[21] J. Silc, B. Robic, and T. Ungerer. Asynchrony in parallel computing: From dataflow to multithreading, 1998.

[22] I. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, 1989.

[23] I. Sutherland. Addressing or function? Technical Report UCIES 2005-is08, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, September 2005.

[24] I. Sutherland. Defining some SHIPs. Technical Report UCIES 2005-is03, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, August 2005.

[25] I. Sutherland. FLEET a one-instruction computer. Technical Report UCIES 2005-is02, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, August 2005.

[26] I. Sutherland. Four views of FLEET. Technical Report UCIES 2005-is12, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, November 2005.

[27] I. Sutherland. Indirection for memory read and write. Technical Report UCIES 2005-is11, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, October 2005.

[28] I. Sutherland. Literals for FLEET. Technical Report UCIES 2005-is07, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, September 2005.

[29] I. Sutherland. Ideas about simulator structure. Technical Report UCIES 2005-is16, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, January 2006.

[30] I. Sutherland. Parallel switch fabrics. Technical Report UCIES 2005-is24, University of California at Berkeley, http://research.cs.berkeley.edu/class/fleet/docs/, July 2006.

[31] I. Sutherland and S. Fairbanks. GasP: A minimal FIFO control. In *ASYNC '01: Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems*, page 46, Washington, DC, USA, 2001. IEEE Computer Society.

[32] A. H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, 1986.

Appendix A

Fleet program for merge sort with 2 channels

```
import src/com/sunlabs/archsim/mike/fleet/fleets/mergeSorter/aliases.fleet


initial codebag START {

move[standing] Memory.ReadCBDOut -> fetcher.In

// initialize the offset register
//move Step.TokenOut -> bitBucket
move (1) -> ChunkSize.In

// setup the offset addresses
move (1) -> OffsetRead.In
// move (100) -> OffsetWrite.In

//move Count.TokenOut -> N.TokenIn
//move N.Out -> Count.In, N.In
//move N.TokenOut -> bitBucket

// load N from the memory (use OffsetStride to prime it)
move (MEM_TO_CHUNK_COUNT) -> Memory.ReadCBDIn

move (0) -> OffsetStride.StartValue
move (1) -> OffsetStride.Step
move (1) -> OffsetStride.StepCount
move (prime_token) -> OffsetStride.Next
move OffsetStride.Counter -> Memory.ReadAddress

// prime Memory.Write interface
move (0) -> Memory.WriteAddress
move (EMPTY) -> Memory.WriteCBDIn

// prime shifter
// ... in MEM_TO_CHUNK_COUNT

// prime compare
move (1) -> Compare.In0
move (1) -> Compare.In1
move (1) -> Compare.In2
move (1) -> Compare.In3
move (prime_token) -> Compare.TokenIn0
```

```
move Compare.Out0 -> bitBucket

// prime Adder
// ... Adder has no pipeline

// prime Subtractor
move oob(1) -> Subtractor.In0
move oob(1) -> Subtractor.In1
move (prime_token) -> Subtractor.NextOut
move Subtractor.Out -> bitBucket

// prime Channnel0CBD
move (EMPTY) -> Channel0CBD.Valid
move (EMPTY) -> Channel0CBD.OOB
move (prime_token) -> Channel0CBD.Select
move (prime_token) -> Channel0CBD.NextOut
move Channel0CBD.Out -> bitBucket

// prime Channnel1CBD
move (EMPTY) -> Channel1CBD.Valid
move (EMPTY) -> Channel1CBD.OOB
move (prime_token) -> Channel1CBD.Select
move (prime_token) -> Channel1CBD.NextOut
move Channel1CBD.Out -> bitBucket

// prime Channel0Stride
//move oob(-100) -> Channel0Stride.Set

// prime Channel1Stride
//move oob(-100) -> Channel1Stride.Set

// prime WriteStride
move (0) -> WriteStride.StartValue
move (1) -> WriteStride.Step
move (0) -> WriteStride.StepCount
move (prime_token) -> WriteStride.Next
move WriteStride.Counter -> bitBucket

// prime GlobalCBD halve chunk count
move (CHECK_FINISHED) -> GlobalCBD.Valid
move (EMPTY) -> GlobalCBD.OOB
move (1) -> GlobalCBD.Select
//move (prime_token) -> GlobalCBD.NextOut
move GlobalCBD.Out -> fetcher.In
}

codebag MEM_TO_CHUNK_COUNT {
move (token) -> Memory.NextReadData
move Memory.ReadData -> N.In,ChunkCount.In,Shifter.In,Memory.WriteData

// the write offset = 2N (number of items + oobs) + 1 (address of N)
//move Shifter.NextIn -> bitBucket
//move Shifter.NextShift -> bitBucket
move (1) -> Shifter.Shift

move (token) -> Shifter.NextOut
move Shifter.Out -> Adder.In1

move (1) -> Adder.In2
move Adder.Out -> OffsetWrite.In,GlobalCBD.NextOut

move Memory.WriteCBDOut -> Rendezvous.In0
```

```
move (CREATE_WRITE_CHANNELS) -> Rendezvous.In1
move Rendezvous.Out -> fetcher.In
}

codebag CREATE_WRITE_CHANNELS {
// output channel
move [standing] Memory.NextWriteSetup -> Sorter0.NextOut,WriteStride.Next,EmptyCBD.Next
move [standing] WriteStride.Counter -> Memory.WriteAddress
move [standing] EmptyCBD.Out -> Memory.WriteCBDIn
move [standing] Memory.WriteCBDOut -> WriteCounter.InToken

// load next chunk to all channels
move (NEXT_CHUNK_TO_CHANNEL_0) -> fetcher.In
move (NEXT_CHUNK_TO_CHANNEL_1) -> fetcher.In
//move (NEXT_CHUNK_TO_CHANNEL_2) -> fetcher.In
//move (NEXT_CHUNK_TO_CHANNEL_3) -> fetcher.In
}

codebag CHECK_FINISHED {
move Compare.TokenOut0 -> ChunkCount.TokenIn
move ChunkCount.TokenOut -> bitBucket
move ChunkCount.Out -> ChunkCount.In,Compare.In0

move Compare.TokenOut1 -> bitBucket
move (1) -> Compare.In1

move Compare.TokenOut2 -> bitBucket
move (NEXT_RUN) -> Compare.In2

move Compare.TokenOut3 -> bitBucket
move (FINALIZE) -> Compare.In3

move (token) -> Compare.TokenIn0
move Compare.Out0 -> fetcher.In
}

codebag NEXT_RUN {
// add 1 to the chunk size to accomodate for the OOB data on the end of each chunk
move ChunkSize.TokenOut -> ChunkSize.TokenIn
move ChunkSize.Out -> Adder.In1,ChunkSize.In
move (1) -> Adder.In2

// setup the read stride
move OffsetStride.Done -> OffsetRead.TokenIn,ChunkCount.TokenIn
move OffsetRead.TokenOut -> bitBucket
move OffsetRead.Out -> OffsetStride.StartValue,OffsetRead.In

move Adder.Out ->  OffsetStride.Step

move ChunkCount.TokenOut -> bitBucket
move ChunkCount.Out -> OffsetStride.StepCount,ChunkCount.In,Rendezvous.In0

move (DOUBLE_SIZE) -> Rendezvous.In1
move Rendezvous.Out -> fetcher.In
}

codebag DOUBLE_SIZE {
move Shifter.NextIn -> ChunkSize.TokenIn
move ChunkSize.Out -> Shifter.In

move Shifter.NextShift -> bitBucket
move (1) -> Shifter.Shift
```

```
move ChunkSize.TokenOut -> Shifter.NextOut
move Shifter.Out -> ChunkSize.In,Rendezvous.In0

move (HALVE_COUNT) -> Rendezvous.In1
move Rendezvous.Out -> fetcher.In
}

codebag HALVE_COUNT {
move Shifter.NextIn -> ChunkCount.TokenIn
move ChunkCount.Out -> Shifter.In

move Shifter.NextShift -> bitBucket
move (-1) -> Shifter.Shift

move ChunkCount.TokenOut -> Shifter.NextOut
move Shifter.Out -> ChunkCount.In,Rendezvous.In0

move (LOAD_WRITE_STRIDE) -> Rendezvous.In1
move Rendezvous.Out -> fetcher.In
}

codebag LOAD_WRITE_STRIDE {
move ChunkCount.TokenOut -> ChunkCount.TokenIn
move ChunkCount.Out -> ChunkCount.In,Adder.In1
move N.TokenOut -> N.TokenIn
move N.Out -> N.In,Adder.In2

// setup the write stride
move WriteStride.Done -> OffsetWrite.TokenIn
move OffsetWrite.TokenOut -> bitBucket
move OffsetWrite.Out -> WriteStride.StartValue,OffsetWrite.In
move (1) -> WriteStride.Step
move Adder.Out -> WriteStride.StepCount,WriteCounter.InCount

// setup write counter and rendezvous
// ... setup by N.Out -> ... above
move WriteCounter.Out0 -> Rendezvous.In0
move (SWAP_OFFSETS) -> Rendezvous.In1
move Rendezvous.Out -> fetcher.In
}

codebag SWAP_OFFSETS {
move OffsetRead.TokenOut -> OffsetWrite.TokenIn
move OffsetWrite.Out -> OffsetRead.In

move OffsetWrite.TokenOut -> OffsetRead.TokenIn
move OffsetRead.Out -> OffsetWrite.In,Rendezvous.In0

move (CHECK_FINISHED) -> Rendezvous.In1
move Rendezvous.Out -> fetcher.In
}

codebag NEXT_CHUNK_TO_CHANNEL_0 {
move (token) -> OffsetStride.Next
move OffsetStride.Counter -> Channel0Stride.Set

// do not mix tokens and data
move Channel0CBD.NextSelect -> bitBucket
move (token) -> Channel0CBD.Select
move (NEXT_DATA_TO_CHANNEL_0) -> Channel0CBD.Valid
move (ERROR) -> Channel0CBD.OOB
```

```
// note here the passive use of the input pipeline
move Channel0Stride.NextSet -> Channel0CBD.NextOut
move Channel0CBD.Out -> fetcher.In

move [standing] Sorter0.NextInA -> Channel0CBD.Select

// setup output channel
move[standing] Sorter0.Out -> Memory.WriteData
}
codebag NEXT_CHUNK_TO_CHANNEL_1 {
move (token) -> OffsetStride.Next
move OffsetStride.Counter -> Channel1Stride.Set

// do not mix tokens and data
move Channel1CBD.NextSelect -> bitBucket
move (token) -> Channel1CBD.Select
move (NEXT_DATA_TO_CHANNEL_1) -> Channel1CBD.Valid
move (ERROR) -> Channel1CBD.OOB
move Channel1Stride.NextSet -> Channel1CBD.NextOut
move Channel1CBD.Out -> fetcher.In

move [standing] Sorter0.NextInB -> Channel1CBD.Select

// setup the output channel
// ... already done for channel0
}

codebag NEXT_DATA_TO_CHANNEL_0 {
move Channel0CBD.NextSelect -> bitBucket
move (MOVE_DATA_TO_CHANNEL_0) -> Channel0CBD.Valid
move (EMPTY) -> Channel0CBD.OOB

move (token) -> Channel0CBD.Select

move Channel0CBD.Out -> Memory.ReadCBDIn
move Channel0Stride.Counter -> Memory.ReadAddress

// claim the token
move Memory.NextReadSetup -> Channel0Stride.NextCounter,Channel0CBD.NextOut
}
codebag NEXT_DATA_TO_CHANNEL_1 {
move Channel1CBD.NextSelect -> bitBucket
move (MOVE_DATA_TO_CHANNEL_1) -> Channel1CBD.Valid
move (EMPTY) -> Channel1CBD.OOB

move (token) -> Channel1CBD.Select

move Channel1CBD.Out -> Memory.ReadCBDIn
move Channel1Stride.Counter -> Memory.ReadAddress

// claim the token
move Memory.NextReadSetup -> Channel1Stride.NextCounter,Channel1CBD.NextOut
}

codebag MOVE_DATA_TO_CHANNEL_0 {
move (token) -> Memory.NextReadData
move Memory.ReadData -> Sorter0.InA

// setup the cbd for channel 0

move (NEXT_DATA_TO_CHANNEL_0) -> Channel0CBD.Valid
move (NEXT_CHUNK_TO_CHANNEL_0) -> Channel0CBD.OOB
```

```
move Channel0CBD.NextSelect -> Channel0CBD.NextOut
move Channel0CBD.Out -> fetcher.In
}
codebag MOVE_DATA_TO_CHANNEL_1 {
move (token) -> Memory.NextReadData
move Memory.ReadData -> Sorter0.InB

// setup the cbd for channel 1

move (NEXT_DATA_TO_CHANNEL_1) -> Channel1CBD.Valid
move (NEXT_CHUNK_TO_CHANNEL_1) -> Channel1CBD.OOB

move Channel1CBD.NextSelect -> Channel1CBD.NextOut
move Channel1CBD.Out -> fetcher.In
}


codebag FINALIZE {
}
codebag ERROR_HALVING {
}
codebag ERROR_DOUBLING {
}
codebag ERROR_DECREASING {
}
codebag EMPTY {
}
codebag ERROR {
}
```