Relocatable Fleet Code

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

UCB-AK08

October 23, 2008

1 Introduction

In this memo, I discuss the hardware requirements necessary for relocatable Fleet [2] code. In a standard computer, code is *relocatable* if its base execution address can be changed. In Fleet, I call code *relocatable* if the set of ships it runs on can be changed.

It is desirable for Fleet code to be relocatable, particularly if the code is dynamically scheduled. In this case, the original set of ships that the code targets may not be available, and it would likely be more efficient to relocate the code to a different set of ships instead of waiting for the original set to be freed.

2 A Simple Program

Consider the following simple program that reads 10 values from memory, computes their sum, and stores the result in a fifo:

```
#ship mem:  Memory
#ship add:  Adder
#ship fifo: Fifo

mem.inAddr.readMany:
  literal 0;                 // 1
  deliver;                   // 2
mem.inCount:
  literal 10;                // 3
  deliver;                   // 4
mem.inStride:
  literal 1;                 // 5
  deliver;                   // 6
mem.outData:
  [10] take, sendto add.in2; // 7
add.in1:
```

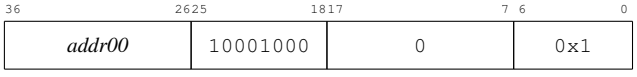
Destination Name	Absolute Address
mem.inAddr.readMany.instr	*addr00*
mem.inCount.instr	*addr01*
mem.inStride.instr	*addr02*
mem.outData.instr	*addr03*
add.in1.instr	*addr04*
add.in1.data	*addr05*
add.in2.instr	*addr06*
add.in2.data	*addr07*
add.inOp.instr	*addr08*
add.out.instr	*addr09*
fifo.in.instr	*addr10*
fifo.in.data	*addr11*

Table 1: Destination addresses in the program.

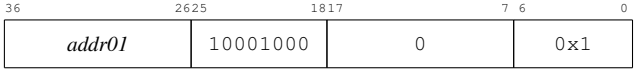
```
  literal 0;                  // 8
  deliver;                    // 9
  [9] take, deliver;          // 10
add.in2:
  [10] take, deliver;         // 11
add.inOp;
  literal Adder.ADD;          // 12
  [10] deliver;               // 13
add.out:
  [9] take, sendto add.in1;   // 14
  take, sendto fifo.in;       // 15
fifo.in:
  take, deliver;              // 16
```

If the Fleet switch fabric uses absolute addressing and assuming the addresses in Table 1, the program will be encoded in memory as follows, as defined by the Fleet architecture manual [1]:

1.


2.


3.

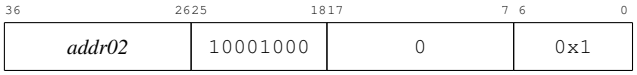

4.


Source Name	Destination Name	Relative Path
mem.outData.data	mem.inAddr.readMany.instr	*path00*
mem.outData.data	mem.inCount.instr	*path01*
mem.outData.data	mem.inStride.instr	*path02*
mem.outData.data	mem.outData.instr	*path03*
mem.outData.data	add.in1.instr	*path04*
add.out.data	add.in1.data	*path05*
mem.outData.data	add.in2.instr	*path06*
mem.outData.data	add.in2.data	*path07*
mem.outData.data	add.inOp.instr	*path08*
mem.outData.data	add.out.instr	*path09*
mem.outData.data	fifo.in.instr	*path10*
add.out.data	fifo.in.data	*path11*

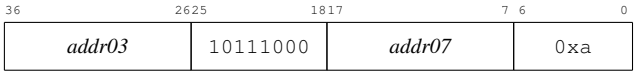
Table 2: Data paths used in the program.

5.

6.

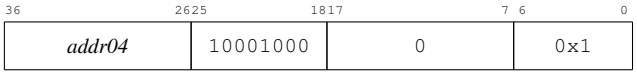


7.

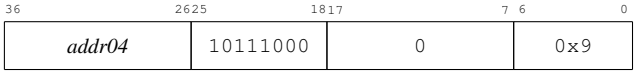


8.

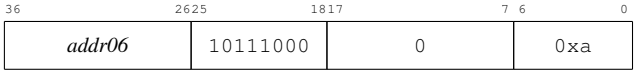
9.



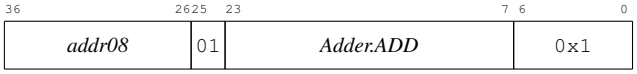
10.

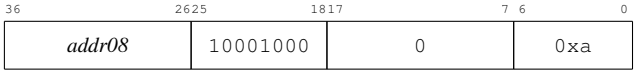


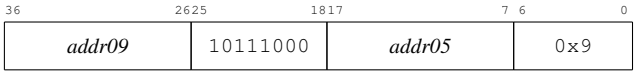
11.

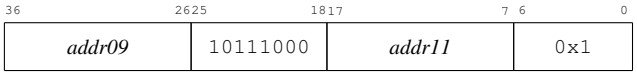


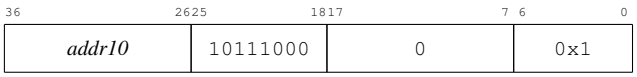
12.



13.


14.


15.


16.


If, on the other hand, the Fleet switch fabric uses relative addressing and assuming the paths in Table 2, the program will be encoded similarly, with *path00* replacing *addr00*, *path01* replacing *addr01*, and so on. I assume that the code is dispatched from `mem.outData`.

3 Rewriting the Code

When moving the code to a different set of ships, each instruction must be modified to run on its corresponding new ship, and each `sendto` instruction must have its data destination replaced. Thus, 19 fields in the above 16 instructions must be replaced.

There are a few cases in which the amount of modification required can be reduced. If the switch fabric uses absolute addressing, and the new set of ships intersects with the old, then the locations in the intersection do not need to be modified. On the other hand, if the switch fabric uses relative addressing, and it just so happens that a relative path in the new set of ships is equivalent to its corresponding old path, then that path need not be changed. In fact, the Fleet hardware and compiler can be arranged such that no relative paths ever need to be modified when moving to a new set of ships.

The Fleet hardware must be divided into groups of ships, which I call *tiles*. Each tile must contain enough of each type of ship such that a Fleet compiler can target any computation for a single tile. All tiles must be composed of the same set of ships, arranged such that all relative paths among them are the same in every tile. Figure 1 shows an example of a Fleet divided into tiles.

A tile is the unit of relocation in a Fleet processor, and a compiler for Fleet must target computation to tiles instead of to arbitrary sets of ships. It is up to the compiler to determine how to divide an entire program among multiple tiles, and the compiler must cooperate with the runtime scheduler to execute the code.

4 Flow Control

According to my description of tiles above, the tiles in a Fleet processor can be arranged in any way as long as the relative locations of each ship are the same in every tile. They can occupy separate portions of the switch fabric, as in Figure 2, or they can be interleaved in the switch fabric, as in Figure 3. The two arrangements, however, have far different ramifications for flow control.

Consider the separated arrangement in Figure 2. Suppose the arithmetic unit (AU) in the red tile needs to send a lot of data to the shift unit (SU), as demonstrated by the thick, red path in the switch fabric. Suppose also that no other

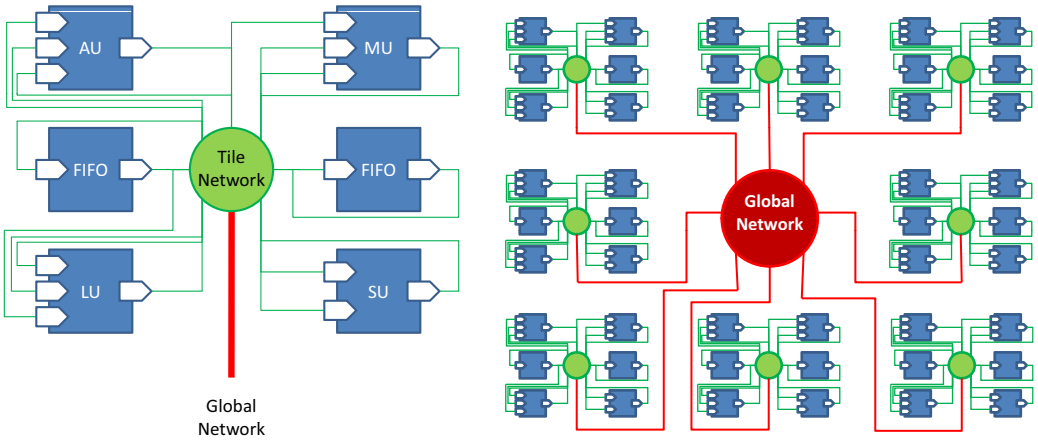


Figure 1: A Fleet processor composed of tiles.

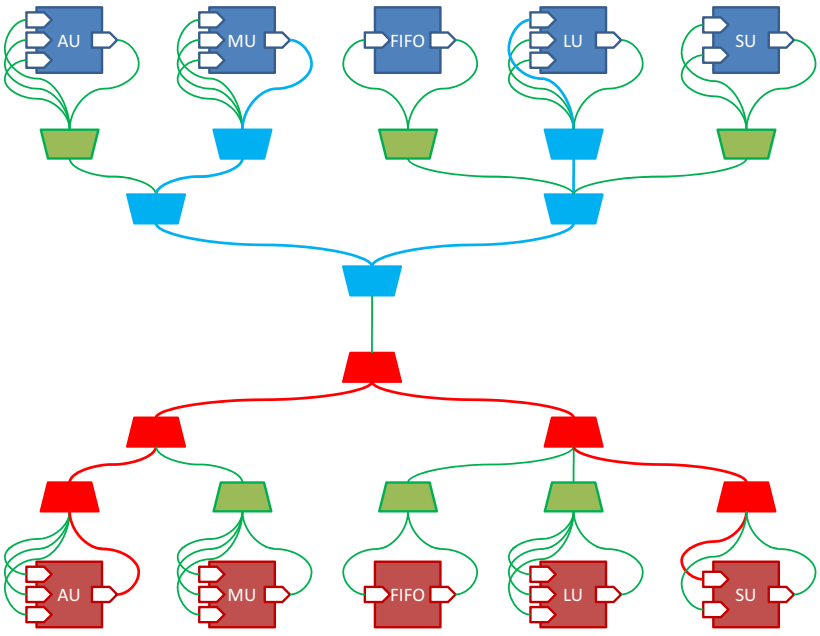


Figure 2: A Fleet processor composed of two tiles in separate parts of the switch fabric.

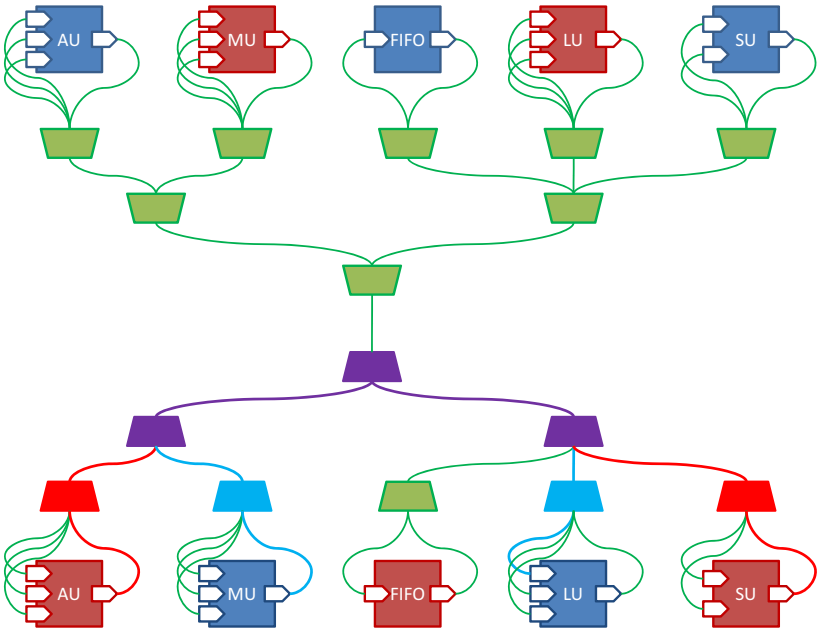


Figure 3: A Fleet processor composed of two tiles interleaved in the switch fabric.

communication is necessary in the red tile. The compiler then may choose to allocate the entire capacity of each link in the red path to the communication between the AU and the SU.

Now suppose that in the blue tile, the only communication required is between the memory unit (MU) and the logic unit (LU), as shown by the blue path. Again, the compiler may allocate the entire capacity of each link in the blue path to this task.

In the separated arrangement, there is no conflict between the communication in each tile. The same operations in the interleaved arrangement of Figure 3, however, result in conflicts for the purple portions of the switch fabric. Allocation of this part of the switch fabric must take into account the communication needs of both tiles. Suppose that the red tile is already running at full network capacity, and a dynamic scheduler wishes to start running code in the blue tile. In the interleaved case, the red tile must be stopped and reconfigured to use less network capacity before the blue tile can start execution. This is not necessary in the separated case.

Thus, in order to minimize the work that the runtime scheduler needs to do, the Fleet processor should be arranged such that tiles are separated from each other in the switch fabric. Communication within a tile should not interfere with any communication outside of the tile.

I assumed in this discussion that data can turn around at any point in the switch fabric. It is only necessary, however, that data travelling between two points in a tile be allowed to turn around at some point before leaving the tile. Thus, a two-level horn and funnel suffices, with shortcuts at the edges of each tile.

5 Conclusion

To summarize, the Fleet processor should obey the following constraints:

1. The switch fabric uses relative addressing.
2. The Fleet processor is divided into sets of equivalent ships, or *tiles*.
3. Each tile has the same relative layout.
4. Communication within a single tile is isolated from communication external to the tile.

If these conditions are met, then the procedure for relocating code is greatly simplified, resulting in simpler compilers and dynamic schedulers.

References

[1] The FleetTwo Architecture Manual, August 2007. http://research.cs.berkeley.edu/fleet/docs/people/adam.megacz/The.FleetTwo.Architecture.Manual.pdf.

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