Goals for the Admiral Language

Amir Kamil
Adam Megacz

UCB-AK03

July 12, 2007

References

[1] Megacz, Adam. *AM14: Syntax*. UCB-AM14.

[2] Kamil, Amir. *MIPS on FLEET*. http://www.cs.berkeley.edu/~kamil/cs294-11/mips.pdf.

1 Introduction

At the current moment, the only ways to program FLEET are using the FLEET assembly language [1] and the MIPS to FLEET translator [2]. Programming in FLEET assembly requires intimate knowledge of the hardware, such as the layout of the SHIPs, type of switch fabric, queue sizes, and so on, and forces the programmer to manually perform tedious tasks such as sequencing. The MIPS to FLEET translator is currently incomplete, and in any case, turns FLEET into a sequential machine.

In this memo, we propose the development of a new language for FLEET that hides some of the architectural details from the programmer. This language can be thought of as the analog of C for FLEET. Because it is used for controlling FLEET, we call the language *Admiral*.

2 Goals

The main goals of Admiral are to provide a measure of independence from the actual architecture of FLEET hardware and to provide some higher-level constructs that would

be tedious for a programmer to implement by hand.

However, we do not intend for Admiral to be a high-level language used by most FLEET programmers. Instead, we expect that Admiral code will be generated by compilers for higher-level languages. As such, we are not concerned with providing constructs such as loops, functions, or even arithmetic operations. At its core, Admiral will still be a language of moves, but they will be more complex moves than those in FLEET assembly.

2.1 Architecture Independence

Though we do not intend to provide complete architectural independence in Admiral, such as insulation from the set of SHIPs contained in the hardware, we do plan on providing some level of independence from the hardware implementation.

2.1.1 Arbitrary Limits

Admiral will allow a programmer to ignore many arbitrary limits in a FLEET implementation, such as the size of instruction FIFOs, the maximum count on a repeating or requeuing instruction, and the size of FIFOs at the leaves of the data horn.

Since performance may suffer considerably in circumventing arbitrary limits, it may be necessary to warn the programmer when limits may be breached, or to provide a profiling tool to report when limits are overrun.

2.1.2 Switch Fabric

The details of the switch fabric will be hidden from the programmer in Admiral, such as its topology and the amount of queue space available. Flow control will be done automatically by Admiral.

2.1.3 BenkoBox Operations

Even though Admiral is still a language of moves, we expect to provide higher-level move operations than are provided by BenkoBoxes. We do not plan on requiring programmers to know what operations (tokenIn, tokenOut, dataIn, dataOut, instruction loops, kills, etc) are provided by BenkoBoxes in order to write Admiral code.

2.1.4 Fetch Units

The operation of the fetch units in FLEET is intimately related to the arbitrary limits and switch fabric in a FLEET implementation, as well as the BenkoBox operations. For example, it is possible to jam the instruction horn by sending too many instructions to a single BenkoBox, so there must be some coordination between BenkoBoxes and the fetch units to prevent this from happening. This coordination will be done automatically by Admiral. In addition, the control flow of an Admiral program will automatically be translated into operations on the fetch units.

2.2 Higher-Level Constructs

In addition to providing a degree of architecture independence, Admiral will also contain some higher-level constructs to make the programmer's job easier.

2.2.1 SHIP Allocator

Admiral will provide a built-in SHIP allocator so the programmer does not have to manually manage them. This can also used to provide an additional level of architecture independence, because the programmer does not have to know the actual number of SHIPs available in the hardware. However, the allocator will not hide the actual type of SHIPs in the hardware, because the user must specify the type of SHIP to allocate.

It is unclear how powerful the SHIP allocator should be. For example, it may be useful to provide a means for allocating sets of SHIPs that are in spatial proximity, to minimize the communication time between them. Another possibility is to virtualize SHIPs, so that the programmer can request more of a particular SHIP than there are in the hardware.

2.2.2 Control Flow

Some control flow constructs will be built into Admiral. These include sequencing, branches, and some form of "codebag goto". However, as mentioned above, loops will not be provided by Admiral.

3 Conclusion

Programming directly in FLEET assembly is difficult, and a higher-level language is clearly needed. We envision Admiral as a step above FLEET assembly, as C is to sequential machines. Other languages above Admiral will also be necessary for programmability, and we expect them to target Admiral so as to avoid dealing with the details of the FLEET hardware implementation.