

AM04: Subroutines in FLEET

Adam Megacz
megacz@cs.berkeley.edu

September 26, 2006

*Programming FLEET is beginning to smell
like writing microcode for the LISP machine.*

– Adam Megacz

1 FLEET-Sharing Considered Harmful

A key assumption throughout this document is that two codebags will never be active in the switch fabric at the same time unless those codebags were specifically designed to share the switch fabric.

Without this assumption, it becomes impossible to even *guess at* whether or not a FLEET will deadlock, or whether or not two codebags will improperly share a SHIP by getting their data mixed up.

2 General Subroutine Invocation: Serial Only

For this reason, a general-purpose subroutine call *does not* need to handle concurrent dispatch of codebags. In the most general case it is simply not safe to concurrently dispatch codebags.

Before an unknown subroutine is invoked, the caller must be sure that the switch fabric is completely clear of instructions, or that those remaining in the switch fabric will evacuate themselves rapidly in a way that cannot impact the behavior of the called codebag¹. The caller may invoke at most one subroutine codebag (parallel invocations are not permitted) and that codebag must evacuate the switch fabric in a similar manner before returning control to the caller.

¹The most conservative formalization requires that the move to the fetch unit be a *causal consequence* of all other datums leaving the switch fabric. Less conservative formalizations are possible, but tricky.

2.1 Invocation Protocol

A subroutine calling convention needs to deal with four concerns:

1. putting arguments where the callee can find them
2. invoking the callee
3. putting the return values where the caller can find them
4. returning control to the caller

Putting arguments where the callee can find them

For a SHIP with n FIFOs, a call with n arguments or less is made by placing each argument in the correspondingly-numbered FIFO. For more than n arguments, place arguments $1 \dots n - 1$ in the FIFOs and place *the memory address* of the remaining arguments in the last FIFO. Allocation of this memory is the caller's concern, although the callee may modify it (the caller cannot assume that it will be returned unmodified).

Invoking the callee

This is the easiest task; simply send the callee code bag to the fetch unit.

Putting the return values where the caller can find them

Return values are passed back to the caller the same way that call arguments are passed to the callee – using the FIFO ships. If more than n values are to be returned, the caller must pass a pointer to a block of memory large enough to hold the excess return values.

Returning control to the caller

The caller should pass in a “successor codebag” to be sent to the fetch unit upon completion. This is passed in as just another argument.

3 Parallel Invocation

It is, of course, desirable to have a mechanism for invoking two “subroutine” codebags in parallel. As previously explained, this is only feasible if these two codebags are “cooperative”. This check can be performed one of two ways:

- statically, in special cases during compilation from a higher-level language (note that this is *not* possible in general)
- dynamically, by some sort of subroutine “header” that indicates which SHIPs it requires (two codebags are cooperative if their required-SHIP-sets are disjoint).

As a side note, in both cases above the “cooperativeness” requirement is transitive: in order for codebags A and B to be cooperative, A must be cooperative with every codebag that B might invoke. This makes the analysis incredibly

complex (or alternatively, incredibly restrictive). For example, any codebag which invoked a virtual method in C++ or a non-static, non-final method in Java would automatically be uncooperative with *every other codebag*.

One of these two mechanisms must be employed. How to implement these is a topic for another memo.

4 Neighbor-FLEET Invocation

I think the ultimate solution is going to be multiple FLEETs on a single chip (ie FLOATILLA). Every invocation of unknown code will take place on a different FLEET. The descriptor of an unknown codebag should *never* be sent to a FLEET's own fetch unit – only to the fetch unit of a neighboring, idle FLEET.

This has one unexpected benefit: a caller can generate arguments of arbitrary size *as the callee consumes them*, without the data ever having to be written to memory. One could imagine the arguments simply “streaming” from the caller FLEET to the callee FLEET. Once the arguments are completely transferred, the codebag in the caller FLEET evacuates all of its instructions and marks the FLEET as available for the next function call.

The essential idea is this: a FLEET is the “atomic unit of control.” Every codebag may assume that it has one entire FLEET to itself. Multitasking must happen either by using multiple FLEETs or by time-multiplexing a single FLEET.

I think this principle brings much more clarity to the following questions:

1. How big should a FLEET be allowed to get before we break it into multiple FLEETs (a floatilla)?
2. How should the FLEETs in a floatilla be interconnected?
3. How do we make the tradeoff between more FLEETs and a larger on-die cache?

In particular, I think a hierarchical floatilla will prove most useful. This will allow the operating system to allocate self-contained subtrees to individual processes (which can then allocate FLEETs to their own threads).