GREG GIBELING
GDGIB@EECS.BERKELEY.EDU
UC BERKELEY
10/24/2005

CS294-4 Homework #6

1.0 Indirection

There are two useful forms of data indirection in a dataflow or process network system such as FLEET: send-indirect and get-indirect. In more common scenarios, these might be referred to as scatter and gather operations.

The fact of the matter is that both forms are useful in a symmetric system such as FLEET, as in most hardware. However the two forms of data indirection can be implemented in the same ways. We should also note that the two forms are both required in a system where data sources are anonymous: otherwise we might easily rid ourselves of get-indirect or the scatter operation by building a facility which implements it given send-indirect.

The need for these operations can be motivated in conjunction with the later discussions in this write-up. Indirection is an efficient and effectively way of encoding dynamic computation structures in a meaningful manner. As such any example with requires a dynamic computation structure will at some point require a type of indirection. Plug-ins of all types are an example of a dynamic computation structure, and yet even more fundamental is a system itself, we must somehow be able to dynamically control the program text which is loaded in a system. The fundamental mechanism for this, even in the case of a standard loader/linker arrangement is a form of indirection.

While this write-up discusses indirection at a lower level of abstraction, machine code versus algorithmic, the power available should be clear. Furthermore indirection represents a convenient way to encode virtualization of resources, a more pressing problem at the hardware level, especially in a concurrent architecture such as FLEET.

2.0 Types of Indirection

In this section we will discuss the three primary abstract implementations of indirection in terms of their behavior in the case of send-indirect. We do this for two reasons: first it should be the significantly more common case and two, this makes the discussion clearer.

Send-indirect in the declarative programming or net-listing style which we have been assuming for FLEET, is a method whereby the destination of a piece of data may be late-bound up to the point when that data must be delivered. The fact that the phrase, late-binding was used should indicate the existence of a naming problem, for a discussion of this see section 3.0 Indirection: Structure & Naming.

It should be noted that send-indirect is very much like self-modifying code, in fact in the case of a net-listing or declarative programming style, send-indirect is exactly half of the requirement for fully self-modifying code, where get-indirect is the other half.

There are three primary abstract implementations of indirection in declarative programming:

- **First class indirection:** In this case the program may specify a message and a destination and some "magical" transport is responsible for delivery. Generating new move instructions to be inserted into the instruction pool is an example of this abstraction.
- **Second class indirection:** In this scheme all messages are considered indirectly sent and received. Second class send-indirect is the equivalent of FLEET as having an extra input port for each output port on each SHIP allowing us to specify the destination of messages originating at that port dynamically. This scheme has the advantage that if it can be efficiently implemented, it is by FAR the most general, making it the most common operation, and the most powerful.
- **Third class indirection:** This case is identical to send-indirect except that the transport is no longer "magic." Demux SHIP based delivery would fall into this category, as would every piece of hardware ever built. The major difference which names this abstraction is the fact that there are first class data-types for addresses or names, instead we are reduced to using programmer invented names or numbers to address demux ports which in the program text are statically connected to other SHIPs. This removes the opportunity for completely dynamic computation structures, but adds in opportunities for meaningful computations to be performed over names.

While in class discussions have, to-date, been concerned almost entirely with either first or third class indirection, it turns out an interesting complication arises in the case that we assume second class send and get indirection, wherein we arrive at a tagged dataflow machine, such as those experimented on by Professor Culler, et al. Naturally the reason for our aversion becomes clear in this context: this is precisely the design FLEET was meant to avoid.

3.0 Indirection: Structure & Naming

Under the current assumptions we have been making about FLEET, the dataflow structure of the program is entirely static. This may include dynamic evaluation pruning in the form of control mux/demux operations as identified by "if" statements of some kind, but the overall structure is static.

In conjunction with this assumption of static structure, we have been assuming that naming of data sources and destinations has been confined to the catalog (namespace) in which a given statement or expression exists. In other words, at the program text level we may directly name only SHIPs, or their ports. At the HDL level, or wherever SHIPs are being specified, we can directly name only the ports of this SHIP. And of course at the algorithmic level, we can name only memory addresses.

It is precisely this set of assumptions: static structure and limited naming which allow us to reason most effectively about algorithms, programs and architecture implementations independently. The abuse or modification of either assumption will significantly conflate the problems of naming, protection, virtualization and analysis (including debugging).

Indirection, of course, is an easy method for violating both the static structure and naming assumptions in one very simple step. Suddenly the dataflow structure of the

computation, is unknowable until execution time. This is precisely because we may in fact use SHIPs which, at compilation time, we were not even aware existed, (in for example a convoluted form of superscalar code rewriting). While this is very powerful indeed, it comes at a severe price in that it makes both the FLEET implementations and programs rather brittle.

4.0 Conclusion

The case of first class indirection where we lend first class status to SHIP port names through the use of special "port address" is, from an abstraction and forward compatibility standpoint, a bad idea. This is primarily because it entails almost all of the complicated support circuitry of second class indirection but without the full integration into the programming model. It has generally been true over the years that partial feature integration in this manner, while clever at first, is expensive and painful as optimizations and features are grafted on later in a system's design. First class indirection does not fully decouple the implementation of indirection from it's abstraction, it mixes the two in a prohibitively complicated way, ensuring that it will be a regrettable design decision in perhaps 5-10 years.

We might solve the abstraction/implementation problem by moving to second class indirection. However this abstraction of indirection suffers the more pressing problem of obscuring the structure of program. Even in cases where the structure is fully static, this scheme demands that it be generated dynamically. Aside from all the reasons which a tagged token researcher could give you, the real reason to avoid this is simply that it introduces the exact problem FLEET was designed to solve: the ISA abstraction should not hide the increasing expense of a move operation. Thus we may write off second class indirection as the exact antithesis of FLEETs reason for existing.

In the end third class indirection is the only tenable solution. At the same time that it restricts us to a static structure, it fully allows for dynamic dataflow. It also represents a complete decoupling of the abstraction and implementation, in that third class indirection can easily be expressed as first class or second class, and may in fact be translated easily using naming tables, and type casting, which in hardware are trivial any may be distributed. Therefore we can always represent our computations with third class indirection, and implement them however we choose, or indeed build any high level construct over them that we choose.

However the reverse is much less true. While the expressive power of all three forms are identical, there exists no constant space and time, distributed structures for the conversion of first class or second class indirection to third class indirection.

Furthermore by allowing naming to be arbitrarily chosen to suit the needs of the programmer, as in third class indirection, we add the opportunity for meaningful operations on names to be defined. For example, in the multiple FIFO matrix corner turn example, computing the next FIFO in a round robin fashion may now consist simply of modulo arithmetic.

Thus, third class indirection, which is merely the use of demux SHIPs, becomes the clear choice now, even as it was before this subject arose. Of course the demux SHIPs could be implemented in the switch fabric by a clever FLEET engineer but either way the optimization choice is not forced on the system designer by the system architecture.