Function Calls for FLEET

Greg Gibeling
Wednesday October 4th, 2006
GDG02 – Function Calls for FLEET

1.0 Introduction

Subroutines or function calls have been a normal part of the programming model for standard ISAs for a very long time. They have been around long enough, in fact, that their original meaning has been somewhat blurred and forgotten.

Function calls are a mechanism for virtualizing the instruction set of a processor. Each new, more complicated, instruction is simply represented by a series of loads to specific input registers, a jump-and-link and a series of calls to restore registers or read output. A function is nothing more than a very complicated machine instruction.

The problem in introducing such a classic function call mechanism to FLEET centers, ironically, around the lack of centralization in FLEET. In a standard ISA, the state of the processor can be virtualized simply by saving and restoring the register file. A FLEET however has far more, far less structured state. State in FLEET exists in each SHIP and even in the switch fabric itself.

Back in Fall of 2005, I produced a document for homework #2 of the CS294-FLEET class which dealt both with function call and the similarity of WaveScalar to FLEET. In this document I will re-present those ideas, and attempt to update them in accordance both with the newer designs for FLEET, and more recent thinking.

2.0 Limited Tags & Backing Store

There are two basic possibilities for virtualization of a FLEET, and thus for implementing function calls. The first, called “stop & swap” relies on a global freeze of all values in the entire FLEET, which is then written to main memory as a block. This mechanism can also, thankfully, be used to implement threads, but it’s use for function calls somewhat complicates the call standard as the arguments and return values must either be in memory, or perhaps the record store is treated specially.

The second mechanism is described in [1] with more data in [2, 3] not to mention [4-6]. I refer to this mechanism as “tagged” as instructions from various threads or functions are given an identity tag, and thus allowed to execute concurrently, but no instructions with one tag may interact with instructions or data with a different tag.

The fundamental computation structure behind function calls is a stack, but the implementations realized with “tagged” and “stop & swap” are both costly. By combining them we can make use of the strengths of each while hiding the weaknesses.

The basic premise is simple: there is a variant of the “load code bag” instruction which marks the incoming code bag as belonging to a new function, one which deserves its own virtual FLEET state. The instruction fetch

SHIP is free to assign any tag to this code bag, meaning there could be anywhere from a minimum of two tags up to hundreds perhaps.

SHIPs and the switch fabric must be extended to handle tags, as in Monsoon and WaveScalar. However, if only two tags are used, there is an added bonus: we can easily limit the complexity of these additions, and in this extreme case only a single bit of tag is needed.

Special moves or a SHIP may also be provided to change the tag of a piece of data, thus allowing it to be passed from one virtual FLEET to another. This mechanism can be used for function arguments and return values, or even for IPC if the tags represent different threads, rather than different stack frames. In truth, the use of 1st class instructions would unify the special "load code bag with new tag" instruction and the ability to change tags on a piece of data.

In order to cope with the limited tag-space, a FLEET must be able to clear all moves and data with the old tag out to some backing store. This can be done with extensions to the edges of the switch fabrics. Moves with the old tag are removed from the various queues and written to memory. Data with the old tag are handled similarly. The major downside to this is the need to engineer SHIPs not to store data internally, otherwise it must be specially handled, and the need to bound the latency on the various switch fabrics so that at some point it is possible to be sure that all old data and moves have been swapped out.

While at first glance this is "stop & swap" with a bunch of needless complicated tags, the benefit is that now the swap operation can take place in the background, and the stop becomes merely a pause. In fact perhaps 3 or 4 tags could be introduced such that the backing store could be transparently spilled and filled to SRAM and then DRAM similar to the register stack implemented by the IA64 architecture.

3.0 Coroutines

Section 2.0 Limited Tags & Backing Store outlines a proposal for using tags and a transparent backing store to implement the virtualization of a FLEET for the purpose on function calls. And yet, that same section also hints at the possibility of expanding the same mechanism to implement coarse grained multi-processing or multi-threading in FLEET.

Namely, by allowing tags to be assigned to functions in such a way that a function has access to both it's predecessor and successor we can easily allow functions to setup arguments and read return values. However, by providing a second mechanism for the assignment of tags, perhaps through some simple data hiding in an OS layer, we can use tags to implement multi-processing.

In contrast to function calls, threads or processes simply have no access to each other's tag-space. This means that each process or thread is completely independent of one another.

Of course relaxing this barrier so that an OS layer could access the tag-space of another process or thread would allow a powerful form of run-time IPC.

In general, this mechanism begins to suggest "co-routines" whereby the distinction between process level and function level virtualization is blurred, sometimes to an extreme. In fact the co-routine model is also well-suited to a FLEET with high concurrency, as coroutines are an inherently concurrent programming model.

The problem with such a co-routine model is that it raises the same issues of traditional dataflow architectures, namely that the scheduling problem is now unbelievably hard, and the memory requirements for virtualization may grow to be very large. Of course the co-routine programming model also provides a solution for this, namely that parallelism is left essentially to the programmer.

This section, unfortunately, contains no true conclusions as the matter deserves more time and thought, not to mention some collaborative exploration.

4.0 Flotillas

As Intel rolls out dual and quad code microprocessors, and the RAMP project begins to investigate 100+ processor systems, the issue of multi-processor FLEET design is one begging for discussion. Because FLEET is designed to be highly concurrent, with very independent SHIPs, it is possible that FLEETs may simply grow very large, and through the use of the virtualization mechanisms in section 2.0 Limited Tags & Backing Store or 3.0 Coroutines, be capable of running multiple functions and processes at one time.

However, the use of tags to separate FLEET contexts also suggests the resurrection of the term "flotilla" from Fall 2005, to describe a collection of FLEETs.

Under the model outlined in above sections a Flotilla might consist of several FLEETs operating in cooperation, with each FLEET taking responsibility for one or more tag-spaces or contexts. This mechanism would mesh well with the use of co-routines as two co-routines could now be executed in parallel on two FLEETs and the tag-changing-moves between them could simply involve the transport of data through some meta-switch-fabric.

Furthermore, this kind of organization could allow a high degree of flexibility in scheduling, a program with many co-routines could be executed using spatial or temporal virtualization, with the various routines on multiple FLEETs at one time, or swapping in and out of one FLEET.

Even better, a Flotilla with this capability could scale the number of threads being executed, as the number of function calls or coroutines changes, thereby trading thread, instruction and function level parallelism all through the same mechanism.

Again, this section offers few conclusions beyond the suggestion and minimal exploration of an idea.

5.0 Conclusion

In general this paper offers few solutions. Section 2.0 Limited Tags & Backing Store provides a rehash and update of an old proposal for the handling of function calls through a mechanism cobbled together from the IA64 transparent backing store and tags [1] or WaveNumbers [4-6].

Sections 3.0 Coroutines and 4.0 Flotillas provide some thoughts on the possible expansion of this tag and swap mechanism to encompass a possible programming model, coroutines, and a multiprocessing model, flotillas.

In the future, I hope the through discussion and perhaps several revisions of this document some additional conclusions and hard research data may be produced.

6.0 References

1. Papadopoulos, G.M. and D.E. Culler. *Monsoon: an explicit token-store architecture*. in *Seattle, WA*. 1990.
2. Culler, D.E. and Arvind. *Resource requirements of dataflow programs*. in *Honolulu, HI*. 1988.
3. Culler, D.E., K.E. Schauser, and T. von Eicken. *Two fundamental limits on dataflow multiprocessing*. in *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism. IFIP WG10.3 Working Conference. Orlando, FL*. 1993.
4. Swanson, S., et al., *Dataflow: The Road Less Complex*. 2003. p. 13.
5. Swanson, S., et al., *Threads on the Cheap: Multithreaded Execution in a WaveCache Processor*. 2004. p. 7.
6. Swanson, S., et al. *WaveScalar*. in *36th International Symposium on Microarchitecture. San Diego, CA*. 2003.