MIPS on FLEET Update

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

May 18, 2007

1 Introduction

The FLEET computer [1] is a new architecture designed from the ground up with parallelism in mind, providing the programmer with direct control over data movement and concurrency. Since FLEET is a completely new architecture, programming for it can be quite challenging. The machine is inherently parallel, so sequential programming techniques do not directly translate to it. The MIPS to FLEET translator [2] addresses this issue by providing a mechanism for running sequential code on FLEET.

Since the MIPS translator was written, FLEET has undergone many revisions, so the translator is not compatible with the current version of FLEET, which is very close to the eventual FLEET 3 hardware implementation. In this report, I discuss the changes necessary to achieve this compatibility. Though the new translator design is close to complete, the implementation still needs significant work before it is finished.

While the MIPS translator allows sequential code to be run on FLEET, it takes little advantage of FLEET's parallelism. In this report, I also discuss one possible scheme for parallelizing the output of the translator.

2 Translator Changes

Since the last semester, both the FLEET architecture itself and the set of SHIPs it is likely to provide have undergone revision. The MIPS translator must be modified to account for both sets of changes.

2.1 FLEET Modifications

FLEET has changed considerably since last semester [7]. *BenkoBoxes* were introduced to replace the data and token inboxes and outboxes [6]. The syntax of FLEET assembly has changed [5], and new instruction forms have been added [4] as well.

Since the MIPS translator does not use many of the features of FLEET, such as counting or standing moves, most changes required by the FLEET updates are relatively straightforward to implement. As a

result, I have not really concerned myself this semester with updating the translator to match the new version of FLEET.

One important complication, however, is the word size of FLEET. This will be handled by either ignoring the extra bits that FLEET provides, or enforcing the invariant that they are always zero. Which option to use depends on the implementation of specific SHIPs such as the Adder SHIP.

2.2 SHIP Modifications

We now have a better idea of what SHIPs will be going into the FLEET 3 implementation. As such, the MIPS translator needs to be modified to target only the SHIPs that will actually be available.

2.2.1 Unsupported Instructions

It is very likely that certain operations such as multiplication, division, and byte-addressed memory will not be provided by the SHIPs in FLEET. Instead of trying to implement these operations in the MIPS translator, I've decided to decrease the compatibility of the translator to MIPS R1000, which does not support these operations either. Thus, it will be the job of the MIPS assembly programmer or compiler to simulate such operations. There is a GCC compiler for MIPS R1000 that implements them in software.

2.2.2 Registers

A BenkoBox without an associated SHIP can be used to implement a register, called a *regbox* [6]. However, it is not clear the FLEET 3 will have regboxes. If it won't, then registers can be implemented as before with FIFOs.

2.2.3 Arithmetic and Logical Operations

The Adder SHIP can be used to provide addition and subtraction, as before. There is now a Lut3 SHIP that can perform bitwise operations on its inputs. This SHIP can be used instead of a separate Logic SHIP to perform basic logical operations such as `and`, `or`, and `xor`.

2.2.4 Shift Operations

During the semester, a BitFifo SHIP proposal was discussed that could be used for performing shift operations, among other things. I have actually implemented such a SHIP.

The purpose of the BitFifo SHIP is to treat a sequence of words as a stream of bits. It accepts whole words as input and contains at least two words worth of storage to keep track of the queued bits. In the current implementation, the BitFifo treats words as little-endian, though it could be modified to treat different input or output words as little or big-endian.

The BitFifo provides commands to extract N bits from the queue, where N is at most the word size, and either drop them or place them at the output. The output is always an entire word, so the rest of the bits need to be filled in. The BitFifo provides separate command forms for filling the extra bits with zeros, ones, or the last bit taken.

In order to use the BitFifo to shift a word right N bits, the following sequence of operations need to be performed:

1. Send the original word to the BitFifo as input.
2. Drop N bits.
3. Take $W - N$ bits, where W is the word size, and fill with zeros for a logical shift or the last bit taken for an arithmetic shift.

In order to use the BitFifo to shift a word left N bits, the following sequence of operations need to be performed:

1. Send 0 to the BitFifo as input.
2. Send the original word to the BitFifo as input.
3. Drop $W - N$ bits.
4. Take W bits.
5. Drop N bits.

In order to avoid using the Adder SHIP to compute $W - N$, the BitFifo SHIP actually provides a separate command form that uses $W - N$ when given N.

2.2.5 Branches and Sets

Branch and set instructions can now be implemented using a combination of the Adder SHIP and a Choice SHIP [3]. The general strategy is to use the Adder to compute the branch or set condition and send the result to the command input of the Choice SHIP. The codebag IDs corresponding to the two branches are sent to the Choice data inputs, and depending on the branch condition, one is routed to the dispatch input of the Memory SHIP while the other is destroyed.

2.2.6 Jumps

The absolute jump instruction requires no modification from last semester. The jump to an address in a register, however, still poses a problem since it depends on the relationship between an instruction address and its associated codebag ID. Most such jumps are used for returning from a function call, which are implemented using either a jump and link (`jal`) or a branch and link. They can thus be handled by constructing a static table mapping the address following each link instruction to its associated codebag, and then performing a lookup at runtime when executing the return instruction. This does not, however, handle target addresses computed by the program that are not from link instructions.

It may actually be preferable to rewrite linked addresses, such as replacing them with their corresponding table index. This would make address lookup simpler and faster.

2.2.7 Memory Access Operations

Since FLEET will not likely provide byte-addressable memory, the MIPS translator will no longer implement the corresponding MIPS instructions. Even the word-addressed instructions in MIPS pose a problem, since word addresses are multiples of the word size in bytes, while they are not in FLEET. Before performing a memory operation, then, the translator must shift the target address by sending it through the BitFifo SHIP.

2.2.8 System Calls

The system calls that can be provided by the translator depend on the input/output specification of FLEET, which has not been worked out yet.

3 Parallelization

During the semester, I have given some thought on how to support instruction-level parallelism (ILP) in the MIPS translator. Since each MIPS instruction is implemented by a sequence of FLEET instructions, it is crucial that parallel MIPS instructions not interfere with each other. This gives rise to a tradeoff between translator complexity and the amount of parallelization that can be achieved.

One simple strategy for parallelization is as follows. Introduce the concept of a *cycle*, as in synchronous machines. The operations in each cycle correspond to a codebag, so they are implicitly parallel, and a cycle is not allowed to begin until the previous one has completed. In each cycle, a SHIP may only be used by a single MIPS instruction. A SHIP is considered to be in use if any of its inputs is occupied by data at the end of a cycle. A cycle is considered to be finished when all data in flight during the cycle have arrived at their respective destinations.

Of course, this description leaves out a lot of details. For example, how should the translator choose which MIPS instructions to run in parallel? How should it decide which SHIPs to allocate to each instruction? How many cycles should it wait before using the output from a particular SHIP? (If the output of a SHIP is used in a particular cycle, then the cycle may have to wait for the SHIP to produce its output, potentially delaying completion of the cycle. So it may be better to wait until a later cycle to use the SHIP's output.) These are all interesting questions to explore, as well as the tradeoffs between different answers.

4 Conclusion

The MIPS translator provides a means of running sequential code on FLEET. With revisions to FLEET and its SHIPs, the translator needs to be updated to continue to provide this service. Much work has been done to this end, but a significant amount still remains.

Though the translator allows an easy way to write code for FLEET, it doesn't provide a way to easily take advantage of FLEET's parallelism. Implementing MIPS instruction-level parallelism in the translator can achieve some degree of parallelism. However, it is likely that in order to truly leverage

FLEET's asynchrony and parallelism, a new programming model needs to be used instead of trying to map synchronous code onto FLEET.

References

[1] The FLEET project. Website: http://research.cs.berkeley.edu/class/fleet.

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

[3] A. Megacz. The choice ship. http://research.cs.berkeley.edu/class/fleet/docs/people/adam.megacz/am17-The.Choice.Ship.pdf.

[4] A. Megacz. Killing and recycling instructions. http://research.cs.berkeley.edu/class/fleet/docs/people/adam.megacz/am15-Decommissioning.and.Recycling.pdf.

[5] A. Megacz. Syntax. http://research.cs.berkeley.edu/class/fleet/docs/people/adam.megacz/am14-Syntax.pdf.

[6] A. Megacz. Unified boxes. http://research.cs.berkeley.edu/class/fleet/docs/people/adam.megacz/am13-Unified.Boxes.pdf.

[7] I. Sutherland, A. Megacz, and I. Benko. Fleet definition. http://research.cs.berkeley.edu/class/fleet/docs/people/ivan.e.sutherland/ies44-Fleet.Definition.pdf.