Channel communications on the Cell Broadband Engine

Damian J. Dimmich
Computing Laboratory
University of Kent
Canterbury, CT2 7NZ
djd20@kent.ac.uk

1. INTRODUCTION

The Cell Broadband Engine[3] (Cell BE) is a processor that can be found in systems such as Sony's Playstation III or IBM's blade servers. The Cell BE consists of a PowerPC[5] core and up to eight SPUs[1], vector processing units, on a single die.

The Transterpreter[4] is a virtual machine for occam-pi [6] which runs on a wide range of devices ranging from small sensor nodes to high-performance clusters. occam-pi provides a consistent set of rich, robust and mathematically backed concurrency primitives which scale well, not only with program complexity, but also with device size. While the Transterpreter already runs on the Cell BE[2], some features of the occam-pi language where not fully supported. This paper describes how one aspect of the language is implemented for the Cell BE.

2. A BIT MORE ON THE CELL BE

Figure 1 approximates the layout Cell BE processor die. The most interesting feature of the Cell BE and what provides most of its performance are the SPU processors. An SPU is a dedicated vector processor with 256kb of local store, and the ability to copy to and from their local store to system memory using a separate memory management unit (MMU). All nine processors and the system memory are interconnected via a high speed bus.

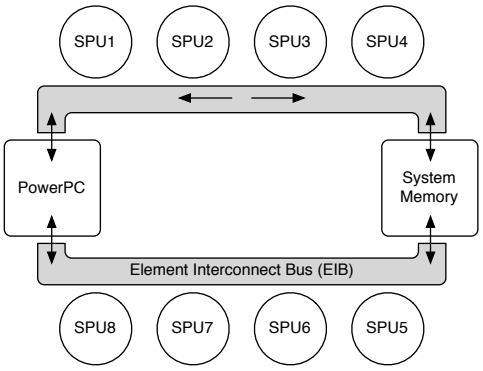

Figure 1: The Cell BE processor.

The MMU allows the processor to transfer a number of queueable 16kb chunks of data between the SPU's local store and system memory. The memory transfers are able to overlap with computation, so while fetching new data to be processed, or sending results, computation on the SPU can continue uninterrupted. Since the amount of local store available on the SPU is limited, it is important to ensure that a constant flow of data is available to each processor so that no processing cycles are wasted by waiting for new data to be fetched.

The result of this novel architecture is a processor that is supposed to be able to deliver around 200 million single precision FLOPS on a single chip. While this level of performance is only attainable by a few algorithms that are well suited to this type of processor, it is still an impressive theoretical peak. The difficulty in leveraging this performance is that a programmer must be able to manage nine memory spaces in parallel while trying to avoid race conditions, deadlock and other problems commonly associated with concurrency.

3. BACKGROUND ON OCCAM-PI

The occam-pi programming language has a notion of channels which are used to synchronise and send data between processes. In order to allow channels to work across processors on the Cell BE, the underlying runtime needs to be aware of how the hardware works and of where a given process is running. Additionally the runtime should transparently overlap data copying to and from system memory and computation.

In order to uphold the semantics of an occam-pi channel, a channel communication must be blocking, unidirectional and not buffered. The implementation of a channel between processors is achieved by a set of helper functions in C that atomically get and set a value in memory that is shared for a given channel. The memory location for this is referred to as the channel word. The channel word, as well as a memory location for storing the data are allocated on startup and kept in system memory.

The Transterpreter runtime has a co-operative scheduler with a FIFO queue, meaning all processes running on it time-share a processor by voluntarily relinquishing control at set points. Each time a process relinquishes control it is sent to the back of the queue. A process can relinquish control by either performing a channel communication or by performing a `RESCHEDULE()` call, which deschedules the current process and adds it to the back of the queue, allowing other processes to run.

4. AN ALGORITHM FOR THE CELL BE

Channel communications across processors on the Cell BE are implemented by compiling in a "magical" set of occam-pi processes into the runtime. These processes provide an interface to the programmer which hides the underlying work that needs to be done in order for a channel communication to take place between processes running on separate processors. The interface exposes one read and one write channel for every other processor in the system allowing a programmer to write programs that can communicate with all processors.

The channel implementation described here relies on the starting processor to allocate channel words and data segments for each channel on startup. All communication occurs via system memory, where data to be sent between processors is first copied by the sending processor from its local store into system memory, and then copied by the reading processor into its local store. Any communication to and from the PowerPC results only in a read or write to system memory as the PowerPC manages system memory directly, and not through an MMU like the SPU.

```
-- checks for new data from 'pid'
-- and forwards it on down the 'in!' channel.
PROC in.chan(CHAN INT in!, VAL INT pid)
  INT tmp:
  INITIAL INT check IS 0:
  WHILE TRUE
    SEQ
      -- Check if data is available
      TVM.read.check(pid, check)
      IF
        check = 1
          SEQ
            -- Fetch the data, send it on, and
            -- reset the channelword to 'empty'
            TVM.read.chan(pid, tmp)
            in ! tmp
            TVM.read.finish(pid)
        -- There is no data, reschedule
        TRUE
          RESCHEDULE()
:
```

Listing 1: The reading process.

Listing 1 shows how the reading end of a channel is implemented for interprocessor communication using helper functions written in C (prefixed with `TVM.`). The `TVM.read.check` function checks the channel word in system memory for a pointer to the channels data. If the channel word contains a null pointer, no data is available for reading and the process issues the `RESCHEDULE()` command, relinquishing the processor for other processes in the queue.

If the channel word contains a non-null value, the data the channel word is pointing to is read into the temporary variable `tmp`, and then sent down the `in` channel. Once the `in` channel has been read the process continues and sets the channel word back to null by calling `TVM.read.finish` signifying that the read is complete.

```
-- out.chan reads the 'out?' channel and
-- copies it to the channel word of 'pid'
PROC out.chan (CHAN INT out?, VAL INT pid)
  INT tmp:
  WHILE TRUE
    -- ?? means that the read from out only
    -- completes at the end of the SEQ block.
    out ?? tmp
      SEQ
        -- Send the data
        TVM.write.chan(pid, tmp)
        INITIAL INT check IS 0:
        WHILE check = 0
          SEQ
            RESCHEDULE()
            -- Set check to 1 on write complete
            TVM.write.check(pid, check)
:
```

Listing 2: The writing process.

Listing 2 shows the implementation of the writing end of a channel. At the start of the outer loop the process blocks (waits) on the `out` channel until another process sends data to it. As soon as data is received `TVM.write.chan` is called, which copies the data in `tmp` to the channel word in main memory. After that it goes into a loop which begins by rescheduling the process to allow other processes to run. This allows some time for the other processor to read the data. When the sending process is scheduled again, it calls `TVM.write.check` to check if the receiving end has completed the data copy and channel communication. A null value in the channel word indicates that the read has completed. If the channel word is null, the variable `check` is set to 1. This causes the inner loop to end and return to the place where the `out` channel is read. Otherwise it reschedules again to allow for some time other processes to run before checking that the read has completed again.

The key to this process is the extended input - `??` - on the `out` channel. This ensures that the read of the `out` channel only completes once code beneath the **`SEQ`** goes out of scope. The process that is sending data to the `out.chan` process will block until the communication with the other SPU has completed, making the communication transparent.

Because of the way occam-pi channels are implemented, a process that has committed to a channel write - `!` - is unable to modify the data it is sending, or back out of the send. This means it is not possible for the language to directly support an "extended output" similar to the extended input. A system which approximates "extended input" can be simulated by using assembly. This is used in an optimised version of the `out.in` process where assembly is used to check if another process is waiting to read on the other end of the `in` channel. Only then does the optimised process start checking system memory for data from another processor. This results in a 30 percent speed up in interprocessor channel communication because of the greatly reduced load on system memory.

5. FUTURE WORK

While channel communication is now supported by the Transterpreter on the Cell BE, a number of other features of the occam-pi language still need to be added. Support for Barriers across multiple processors and automatic deadlock detection will be added and algorithms to implement those need to be devised. Furthermore, channel communication can be optimised by making use of additional hardware features available on the Cell BE.

Direct support for the Cell BE could be added to the occam-pi compiler by making it aware of the processors architecture. This would allow for arbitrary numbers of channels between processors. This would help remove the current restriction of just one in and one out channel between any two processors.

www.transterpreter.org

6. REFERENCES

[1] B. Flachs, et al. A Streaming Processing Unit for a CELL Processor. *ISSCC - Digest of technical papers*, 2005.

[2] D. J. Dimmich, et al. A Cell Transterpreter. In P. Welch, J. Kerridge, and F. Barnes, editors, *CPA*. IOS Press, 2006.

[3] D. Pham, et al. The Design and Implementation of a First-Generation CELL Processor. pages 184–185. IEEE ISSCC, February 2005.

[4] C. L. Jacobsen and M. C. Jadud. The Transterpreter: A Transputer Interpreter. In *CPA*, pages 99–107, 2004.

[5] R. Kalla, et al. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40– 47, 2004.

[6] P. Welch and F. Barnes. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, 2005.