The Challenges of Writing Software for Massively Parallel Architectures.

Jason McGuiness, Colin Egan, Bruce Christianson

28th June 2005

University of Hertfordshire, Hatfield, Hertfordshire, U.K. AL10 9AB

c.egan@herts.ac.uk
Introduction.

- Background - a rapid overview of cellular architectures.
- A state of the art: Cyclops - developing programs for it.
- A review of the problems faced developing for Cyclops.
- Some suggested solutions and future directions.
Background.

Some key properties of cellular architectures:

- A key feature: overcome the memory wall by intimately associating memory and thread units on the same chip die.
  - This reduces memory-access latencies,
  - but this reduces processor throughput, because of design constraints, the execution units would be more simple than super-scalars.

- This reduced processor-throughput would be offset by designing the CPUs to be interconnected into massive arrays.
  - Of the order of 64 execution units per die, scaling to an array of interconnected computers with over 10 million thread-units.
A more detailed view:

The key features of a cellular architecture are:

• In the order of 10 million thread units, implies:
  – It would be vital to have effective, or even efficient, techniques to address the available parallelism. For example, cheap thread creation, destruction and synchronisation.

• As a direct result of such high parallelism:
  – Memory bandwidth would be issue: but the proximity of memory to thread units mitigates against.
  – Software-controlled data caches further help.
  – Effective hardware support of shared data would be vital to reduce contention and stalls.

• A number of high-bandwidth interconnects to the other, off-chip, thread units and memory.
The Cyclops implementation: a rapid overview:

• This has a memory hierarchy different to super-scalars: the different levels support different consistency models:
  – The level closest to a thread unit, the “scratch-pad”, maintains sequential consistency memory-accesses for that thread unit. For other thread units it maintains program consistency.

• The next on-chip level, maintains sequential-consistency accesses for that memory address (or possibly bank that contains it) for all contending thread units, and program consistency otherwise.

• For off-chip memory accesses, a message-passing protocol is implemented in hardware that supports asynchronous, queued, memory requests using a large cross-bar.
The software perspective:

To effectively use cellular architectures requires software support, that would consist of:

- A run-time system that
  - Efficiently supports the threading model: i.e. cheap to both create and destroy.
  - Supports efficient synchronisation objects, supporting the different memory-consistency models, both on and off-chip: for both the consistency models and the message-passing model.

- Supports effective inter-chip data-migration.

- Supports program start-up, shut-down and “restore points”.
Details regarding the Cyclops threading model:

The initial threading model (C-threads) was based upon the POSIX pthread model:

- This is a low-level, complex API, replete with threading primitives:
  - It is easy to implement upon the Cyclops ISA, and is relatively well-known.
  - Unfortunately, I contend it is not conducive to writing multi-threaded programs, let alone massively parallel ones, leaving correctness questions aside.

- For example, simply synchronising data structures between threads, and also ensuring correctness is hard. This combined with a sophisticated memory-model further compounds the programming problems, due to the non-uniform techniques of accessing shared data.

- The POSIX pthread library mixes threading and synchronisation primitives, thus concepts, into the same library, which does not help.
Problems with writing multi-thread programs.

It is common lore in computing that writing multi-threaded programs is hard: perhaps the language we use to express such programs contributes to this:

- Mixing synchronisation and threading objects into the same library confuses the API:
  - These are complimentary, but separate, language-level concepts.
  - Thread creation and thread joining are a separate concept from synchronisation objects such as mutexes or condition variables. e.g. POSIX.

- Managing access to shared data must be done explicitly by the programmer. On Cyclops they must choose, guided by the memory location of the data, which synchronisation primitive, if any, is required:
  - Off-chip access is made by arbitrary thread-unit(s) to arbitrary, off-chip memory location(s).
  - Access to on-chip memory is sequential- or program-consistency depending upon the thread unit and memory location.
A brief introduction to Cyclops programming:

I developed a “proof of concept” program for a prototype version of Cyclops:

- It had to be sufficiently large and have irregular synchronisations to test the threading, synchronisation, memory models and locking primitives.

- I chose an implementation of a Mandelbrot generator utilising a work-stealing algorithm. The work-stealing operated between some of the thread units, requiring both shared and unshared scratch-pad based data, and shared global-data.

- The compiler placed:
  - All local, automatic, variables (i.e. the stack frame) in the scratch-pad associated with the thread-unit of that running thread.
  - All static variables were placed into on-chip memory, in principle, obeying sequential consistency. (Unfortunately, due to hardware bugs at the time, this was not the case.)
Cyclops library support.

- Based upon the C-thread API (which is similar to the POSIX API) I wrote:
  - A more uniform thread-library abstraction to ease creating threads and, more importantly, managing their state, so that thread-local storage was also automatically placed into the scratch-pad, without resorting to C-thread primitives.
  - A synchronisation library that expressed, in a more convenient way, the synchronisation needs of the Mandelbrot program. This meant, for example, that locking scratch-pad or on-chip memory located shared data was done using the same programmatic constructs, thus allowing the compiler to choose the most efficient primitives.

- The above two points are key: I contest that there is an open question regarding writing multi-thread programs and finding a language in which it is conducive for humans to write such multi-threaded programs. (I do not claim to have solved this!)
The Mandelbrot program: A critique.

The library support that was written has problems:

• It was too focused towards heavy-weight threading. i.e. poorly expressed creating or destroying threads on the fly.

  — This limitation was due to poor abstraction in the library. I contend that the concept that a thread is an explicit object to be managed is misguided.

• The locking poorly mapped on to the features of Cyclops.

  — This was due to multiple reasons: the data structures in the program were too large, and non-atomic, so the native, atomic, word-sized, memory-operations could not be used. This meant that a key feature of Cyclops could not be exercised: the automatic locking of word-sized memory operations via the various memory-models. (Hardware bugs did not help.)

• Due to hardware problems, accurate execution timings were unavailable, and hardware limitations prevented taking scalability measurements.
Cellular architectures: pushing the boundaries:

The combination of the massively parallel nature and the memory-consistency models severely push the boundaries of current software support at all levels, the library, compiler and tool-set:

- Both threading and synchronisation libraries inadequately aid the programmer in their struggle to create multi-threaded programs.

- Thread generation: should this be done by the compiler, for example expressed indirectly using UPC or HPF, or more directly by the programmer using a thread and synchronisation library such as OpenMP?

- Inter-chip communication: how should this be optimised? By the programmer or by the compiler, for example Tang’s work on EARTH or in the hardware, such as micro-threading?

- Program I/O, particularly with respect to debugging (assuming that the program cannot be proven to be correct) is an open question. How would mitigation against the implicit serialization that debugging implies be done?