Barrier Synchronisation for occam-pi

Frederick R.M. Barnes, Peter H. Welch and Adam T. Sampson

***Abstract*—This paper introduces a safe language binding for CSP multiway events (*barriers*) that has been built into occam-π (an extension of the classical occam language with dynamic parallelism, mobile processes and mobile channels). Barriers provide a simple way for synchronising multiple processes and are the fundamental control mechanism underlying both CSP (*Communicating Sequential Processes*) and BSP (*Bulk Synchronous Parallelism*). The occam-π barriers are more general than those of BSP (an occam-π system can contain any number of barriers, with some processes ignoring them and some registered with many). On the other hand, they are also, currently, less general than those of CSP (occam-π processes must *commit* to barrier synchronisation — it cannot be used as part of a *choice* or ALT). Structured support for *resignation*, a higher-level CSP design pattern, is also built into occam-π barriers. Applications are outlined for fine-grained modelling of dynamic systems, where the barriers are used for maintaining simulation *time* and synchronising safe access to shared data between millions of processes. Implementation details and early performance benchmarks (16 nanoseconds per process per barrier synchronisation on a 3.2 GHz. Pentium IV) are also presented, along with some likely directions for future research.**

***Index Terms*—concurrency, synchronisation, barriers, occam-pi, CSP, simulation, fine-grained, dynamics, time**

I. Introduction and Motivation

THIS paper describes the addition of multiway *barrier* synchronisation to the KRoC [1], [2] occam-π system. The occam-π programming language [3], [4], [5] is a modern version of classical occam [6], including features such as data, channel and process mobility (taken from Milner's π-calculus[7]), dynamic parallelism, extended rendezvous and process priority.

Barriers are a synchronisation primitive on which parallel processes *enroll*, *synchronise* and *resign*. When a process synchronises on a barrier, it is blocked until all other processes enrolled on the barrier have also synchronised. Once the barrier is completed, all blocked processes are rescheduled. The semantics of barrier synchronisation are exactly those of an *event* in *Communicating Sequential Processes* (CSP) [8], [9]. The occam-π language binding is *safe* in the sense that enrollment and resignation are automatically coordinated and that a process may synchronise on a barrier if, and only if, it is enrolled.

Barriers are used for a variety of purposes and with varying granularity in parallel programs. For example, the *Bulk Synchronous Parallelism* (BSP) [10] model describes parallel processes that run (mostly) independently on separate processors, but periodically synchronise on a single global barrier to exchange data. Such models will be supported by the networked version of occam-π (not yet released [11]). In this paper, we are concerned with much finer levels of control, with processes enrolling, synchronising and resigning dynamically on multiple barriers. We are particularly interested in applying these mechanisms to the design and implementation of highly dynamic systems, where the barriers may be used to maintain global and/or localised models of time and to synchronise safe access to shared data (without the need for more expensive locking primitives).

F.R.M. Barnes, P.H. Welch and A.T. Sampson are members of the Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, England. email: {frmb,phw,ats1}@kent.ac.uk

A previous implementation of barriers in KRoC [12] provided *user-defined* abstract data types [13]. '`BARRIER`' variables could be declared, explicitly flagged as *shared* (through the use of compiler directives which overrode parallel usage checks) and operated via a number of procedure-calls ('`initialise.barrier`', '`synchronise.barrier`', etc.) implemented in ETC (*Extended Transputer Code* [14]) assembler. This was functional and fast, but the programmer had to ensure that barriers were initialised correctly, that only enrolled processes could synchronise or resign and that barriers were not assigned or communicated (the semantics of which were undefined).

In the language binding presented here, barriers are declared in the same way as ordinary variables and channels. These barriers are *fixed*, however — they may not be communicated or assigned, but may be renamed (e.g. through parameter passing and abbreviation). Any process that declares a barrier is automatically enrolled on that barrier, and only processes enrolled on a barrier may synchronise on it. If an enrolled process itself goes parallel, the default semantics are that only one of its sub-processes inherits the enrollment — this is checked at compile time. However, an enrolled process may enroll *all* parallel sub-processes on its barrier(s), by explicitly declaring this at the relevant `PAR`.

An enrolled process may temporarily *resign* from a barrier — crucial for the *'lazy'* execution of simulation processes that have nothing to do for long periods of *'time'* — but its re-enrollment is automatic at the end of an explicit `RESIGN` block. An enrolled process automatically resigns from its barrier when it terminates, so that other processes may continue to use it. The semantics of *resignation* are not directly given by CSP, however they are easily modelled by the instantiation of a proxy process that repeatedly offers to synchronise (on the resigned barrier) until the event signalling the end of the resignation happens — see the end of Section II (C).

Finally, we note that although these barriers are *static* entities — like classical occam channels — occam-π offers *mobile* channels and, so, *mobile* barriers are a natural and necessary extension that will be considered in the future.

The language binding of these barriers in the KRoC occam-π system is covered in Section II. Section III describes the implementation. Application techniques are discussed in Section IV. Some early conclusions, including initial performance figures, are given in Section V along with a discussion of future work.

II. Language Binding

Barriers are declared in the same way as ordinary channel and variables, with the process following the declaration automatically enrolled. For example:

```
BARRIER b:         -- declaration of 'b'
...  process(es) synchronising on 'b'
```

To enroll all sub-processes on a barrier, the parallel composition must explicitly declare this. For example:

```
PAR BARRIER b
  P (b)            -- all these
  Q (b)            -- sub-processes
  R (b)            -- are enrolled on 'b'
```

A replicated parallel may also enroll its sub-processes:

```
PAR i = 0 FOR n BARRIER b
  worker (i, b)   -- all enrolled on 'b'
```

In network diagrams, we represent a barrier as a 'bar', connected to all enrolled processes. Figure 1 shows the process network for the above 'worker' fragment.

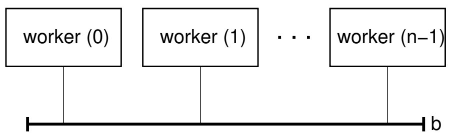

Fig. 1. Barrier synchronised worker processes

Barrier synchronisation is expressed through a new SYNC primitive. For example:

```
PROC worker (VAL INT id, BARRIER x)
  SEQ
    ...  computation
    SYNC x
    ...  more computation
:
```

The execution of the above SYNC line blocks until *all* other processes enrolled on the barrier similarly SYNC.

Note that if a process has a barrier *parameter*, any invocation must have passed a barrier *argument* on which the invoking process was enrolled. Hence, we (and the compiler) may assume that the code body of the process with a barrier parameter *is* enrolled on the barrier and that it is legal to synchronise.

An enrolled process that goes parallel in the normal way (i.e. without explicitly referencing its barrier) passes its enrollment to *only one* of its sub-processes. For example:

```
PROC worker (VAL INT id, BARRIER x)
  PAR
    A ()         -- not enrolled
    B (x)        -- enrolled on 'x'
    C ()         -- not enrolled
:
```

For a normal non-enrolling PAR such as this, exactly which of its sub-processes takes the enrollment does not matter. The compiler checks that only one does though.

An enrolled process may temporarily *resign* from a barrier through the use of a RESIGN-block. For example:

```
PROC worker (VAL INT id, BARRIER x)
  SEQ
    P (x)      -- enrolled on 'x'
    RESIGN x
      A ()     -- not enrolled on 'x'
    R (x)      -- enrolled on 'x'
:
```

Whilst executing process 'P(x)', this 'worker' must synchronise on the barrier (or it will block other enrolled processes that *are* synchronising). However, whilst executing the RESIGN-block 'A()', it plays no part in the barrier and other enrolled processes can synchronise amongst themselves freely. After the RESIGN-block, it is back in the barrier.

Note that some care must be taken to avoid nondeterminism after exit from a RESIGN-block, since the precise time of that exit *and consequent re-enrollment in the barrier* is scheduling dependent. This is considered further in Section II.C below.

A. Barrier usage rules

Process *enrollment* on a barrier is determined by the scope of its declaration, PAR BARRIER compositions and RESIGN blocks. The following usage rules for barriers are enforced by compiler checks:

- a process may only SYNC on a barrier to which it is *enrolled*;
- when resigned from a barrier, a process may not make any reference to that barrier (e.g. to pass it on as an argument to a procedure);
- *only one* parallel sub-process of a non-enrolling PAR may refer to a barrier enrolled at the start of the PAR (e.g. to SYNC on it or pass it to a procedure).
- an individual barrier may be passed to *only one* parameter of a PROC. Strict anti-aliasing laws apply: *different* barrier names always refer to *different* barriers.

The following shows an illegal fragment of occam-π code, that attempts to use a barrier in parallel without extension:

```
PAR BARRIER b
  SYNC b
  PAR
    P (b)   -- error: only one of these
    Q (b)   -- sub-processes may use 'b'
```

The compiler will report a suitable error message, indicating that the inner PAR uses 'b' in parallel without extending it.

B. Multiple barriers

We may enroll multiple barriers in the same PAR construct. In the following example, the 'timer' processes controls the timing of 'process.a' and 'process.b' by synchronising on their respective barriers regularly (at 'long' or 'short' time intervals). Processes 'process.a' and 'process.b' (which may resign from either or both time-slicing controls from time to time) also use a private barrier, 'b', to synchronise between themselves:

```
BARRIER long, short:
PAR BARRIER long, short
  PAR
    long.timer (long)
    short.timer (short)
  BARRIER b:
  PAR BARRIER b, long, short
    process.a (long, short, b)
    process.b (long, short, b)
```

C. Deadlock

The use of barriers in occam-π programs introduces new opportunity for deadlock, caused by incorrect process synchronisation. Consider the processes:

```
PAR BARRIER a, b        PAR BARRIER a, b
  SEQ                     SEQ
    SYNC a                  SYNC a
    SYNC b                  SYNC b
  SEQ                     SEQ
    SYNC b                  SYNC a
    SYNC a                  SYNC b
```

The system on the left, above, deadlocks immediately. Its first process initially offers the 'a' event and refuses 'b'. Its second process does the reverse. This is equivalent to STOP. The system on the right synchronises smoothly and terminates — it is equivalent to SKIP.

Such deadlocks are fairly obvious, however! A more subtle problem can arise through careless use of RESIGN blocks:

```
PROC always (BARRIER a, b)
  WHILE TRUE
    SEQ
      SYNC a
      ...  phase A compute (no SYNCs)
      SYNC b
      ...  phase B compute (no SYNCs)
:

PROC sometimes (BARRIER a, b)
  WHILE TRUE
    SEQ
      SYNC a
      ...  phase A compute (no SYNCs)
      SYNC b
      ...  phase B compute (no SYNCs)
      IF
        ...  decide on a hoiliday
          RESIGN a, b
            ...  enjoy holiday (e.g. sleep)
        TRUE
          SKIP
:

PAR BARRIER a, b
  always (a, b)
  sometimes (a, b)
```

So long as 'sometimes' stays enrolled in its barriers, all goes well — 'sometimes' and 'always' will continue their respective phased computations in parallel, keeping in step with each other as each phase ends.

If 'sometimes' decides to go on holiday, it resigns from its barriers and does other things (like sleep), leaving 'always' to continue on its own — all is still well.

The problem arises if 'sometimes' decides to come back. When it exits its RESIGN block, it re-enrolls on its barriers and waits to SYNC on 'a'. If 'always' is in its phase B when this happens, we are lucky and the two processes resume in perfect synchronisation. But if 'always' is in phase A, its next SYNC is on 'b' and the system will deadlock.

To do this safely, 'sometimes' must coordinate its return with 'always'. One way to do this is for 'sometimes' to request permission from 'always' to return to their joint computations. It must do this before exiting its RESIGN block. The 'always' process only grants this permission in its phase B and, then, waits for confirmation from 'sometimes' that it has re-enrolled (i.e. has left its RESIGN block).

This behaviour is easy to manage by signalling and polling over standard channels:

```
PROC sometimes (BARRIER a, b, CHAN BOOL sig!)
  WHILE TRUE
    SEQ
      SYNC a
      ...  phase A compute (no SYNCs)
      SYNC b
      ...  phase B compute (no SYNCs)
      IF
        ...  decide on a hoiliday
          SEQ
            RESIGN a, b
              SEQ
                ...  enjoy holiday
                sig ! TRUE   -- request
            sig ! TRUE       -- confirm
        TRUE
          SKIP
:

PROC always (BARRIER a, b, CHAN BOOL sig?)
  WHILE TRUE
    SEQ
      SYNC a
      ...  phase A compute (no SYNCs)
      SYNC b
      ...  phase B compute (no SYNCs)
      PRI ALT
        BOOL any:
        sig ? any     -- grant comeback
          sig ? any   -- wait for confirm
        SKIP
          SKIP
:
```

and where the system is now:

```
CHAN BOOL sig:
PAR BARRIER a, b
  always (a, b, sig?)
  sometimes (a, b, sig!)
```

In a larger system, there may be many processes, like '`sometimes`', that retire from the computation from time to time. Examples arise in large scale simulations of dynamic systems, where not all processes need to be continually active (because nothing is changing in their neighbourhood) but need to rejoin some barrier synchronisation (e.g. for managing simulation 'time') when something happens close to them.

In such cases, the above *comeback/confirm* protocol is needed between each resigning process and just *one* specialised process, like the above '`always`', that is *always* cycling and synchronising (and which need do nothing else). Separate *comeback* and *confirm* channels will be needed, `SHARED` at the resigning process ends.

Finally, we note that a process enrolled in some barriers may terminate, at any time, without deadlocking other enrolled processes that remain. As mentioned in Section I, a terminating process automatically resigns from any barriers on which it is enrolled. This is neatly illustrated by the following examples, which demonstrate that the `SKIP` is a *unit* of all occam-π versions of the `PAR` operator:

```
PAR          =     PAR BARRIER b     =     P (b)
  P (b)              P (b)
  SKIP               SKIP
```

In the first system, '`b`' must be a global barrier and `SKIP` is not enrolled. Hence, `SKIP`'s existence and termination have no impact on the continuing operation of '`P(b)`'.

In the second system, `SKIP` *is* enrolled on '`b`'. If '`P(b)`' synchronises on '`b`' before `SKIP` terminates, it simply blocks until `SKIP` does terminate (which is the first and only thing it ever does). But termination is atomic with resignation from the barrier, allowing '`P(b)`' to proceed. If `SKIP` terminates before any synchronisations from '`P(b)`', it has resigned from the barrier and future synchronisations will not be blocked. Either way, the `SKIP` has no impact and we are left with '`P(b)`'.

The resignation from barriers of terminating processes is not directly given by standard CSP, but it is easy to model. Details will be presented in a later paper.

III. Implementation

The implementation of occam-π barriers follows the structures and logic described in [12]. Memory overheads are particularly lightweight, requiring only 4 words of memory for the barrier data-structure. This structure comprises of a count of the number of enrolled processes, the number of *yet-to-*`SYNC` processes, and two queue-pointers holding the list of processes that are blocked *trying-to-*`SYNC`. When the *last* process tries to `SYNC`, the blocked process queue is simply appended to the run queue — a unit time operation, regardless of the number of processes being released.

The implementation is assisted by four new ETC (*Extended Transputer Code* [14]) instructions:

- initialise a barrier (with no enrolled processes).
- enroll n processes on a barrier.
- resign n processes from a barrier, that may cause a reschedule if the barrier is completed.
- synchronise on the barrier, that may cause a reschedule if the barrier is completed.

The ETC to native-code translator in KRoC generates single blocks of in-line target assembly for each of these instructions.

The code generated by the compiler for handling barriers is simple. When a new barrier comes into scope, it is initialised. When a barrier-*enrolling* `PAR` with n sub-processes is entered, a further $n-1$ sub-processes are enrolled — the process executing this construct must already be enrolled on the barrier and takes over one of its sub-processes.

When each barrier-*enrolling* `PAR` sub-process terminates, it is resigned from the barrier. An exception is made for the *last* sub-process that terminates and it is *not* resigned. So, the number of processes enrolled on the barrier before and after a barrier-*enrolling* `PAR` remains the same.

An ordinary `PAR` construct entered by a barrier enrolled process requires no special code generation — only the compiler checks on correct useage (i.e. that at most one sub-process synchronises on the barrier).

When entering a `RESIGN` block, the compiler generates code to resign the process from the barrier (which could result in barrier completion). When the resign block completes, the process is re-enrolled.

Note: no information about the *actual* processes enrolled is held in the barrier data structure — only *how many* there are. Compiler usage checks ensure that only enrolled processes may synchronise or resign — hence, no run-time checks are needed to ensure safety.

Currently, no consideration is given to process priority [5], although this could be added with relative ease. A design constraint, therefore, is that all processes synchronising on a barrier must have the same priority. This restriction permits a very efficient implementation, the performance of which is examined in Section V.

Efficiency is further traded for some flexibility in the current implementation. Barrier synchronisations may not be used as guards in a choice (`ALT`). It would be fairly easy to allow *just one* of the enrolled processes to `ALT` on the barrier, with only a slight loss in efficiency. Allowing *all* enrolled processes to `ALT` on the barrier, however, requires a *referee* process and a *two phase commit* protocol [15], [16]. This is expensive — compared to committed barrier synchronisation — and is left for later consideration.

IV. Barriers for shared data and time

Shared data is not normally allowed between parallel processes in occam-π. However, in the interests of simplicity and performance, there are circumstances when it makes sense. Access to such shared data must be strictly coordinated to avoid race hazards. Barriers provide a highly efficient way to do this but, for now, responsibility for correct management lies with the programmer.

This section presents a design-rule for one way of correct management that is applicable to fine-grained parallel simulations of dynamic systems. The sharing of data must follow a regular pattern and be strictly CREW: either multiple processes are reading some shared data (*Concurrent Read*), or a single process is modifying it (*Exclusive Write*).

A user-defined '`CREW`' abstract data type, providing efficient and correct locking procedures suitable for arbitrary patterns of use, has been available in occam-π [12] for some time. However, barriers are substantially simpler and faster than these general CREW locks and are, therefore, to be preferred when the usage pattern is regular.

In this paper, we are concerned with processes sharing data in a same-memory environment. As we know from BSP models for parallel computing, barriers are an excellent means for coordinating regular distributed computations. Distributed shared memory requires additional care for efficient management (e.g. *PastSet* [17] and numerous implementations of BSP). Extending occam-π barriers across distributed systems is deferred for later work.

A. Visualisation and termination of a cellular automaton

As an example of how barriers can be used both to protect shared data and maintain simulation time, Figure 2 shows a system consisting of a pipeline of 'cell' processes and a single 'display' process. Every simulation time unit, the cells update *their own parts* of some externally visible 'state' — with the display process safely observing and rendering it for visualisation. The cells and display process also share a boolean 'ok' flag, used to signal termination.

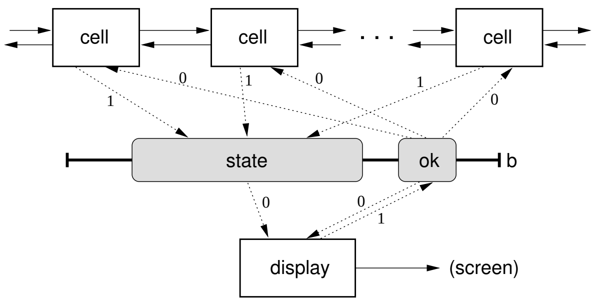

Fig. 2. Shared data for visualisation

CREW rules require that while one or more processes are reading shared data (e.g. cell states by the display, the 'ok' flag by the cells), no process is writing to that shared data (i.e. cell states by the cells, the 'ok' flag by the display). Also, of course, no writes to the same piece of shared data may take place at the same time.

These rules are enforced by dividing the execution cycle of the cell and display processes into two *phases*, with barrier synchronisation implementing the division. Figure 2 extends the symbology of Figure 1. The shaded rounded boxes represent state variables, shared by the cell and display processes. They are stuck on the barrier, `b`, to indicate that access to them is controlled through the barrier. The dotted arrows between the processes and the shared variables indicate two things: reading or writing (depending on the arrow direction) and that the processes must synchronise on the underlying barrier to coordinate that reading or writing. The numbers annotating the read/write arrows indicate the phases in which the reading or writing takes place.

To check CREW conformance, we just have to check that no read/write or write/write on shared state happens in the same phase. In this system, that is trivial, since reads only happen in phase 0 and writes in phase 1. [Note: The parallel writes to 'state', happening in phase 1, are to separate parts of that 'state'.]

The code outline for the overall network in Figure 2, assuming some constant '`n.cells`' is:

```
...  declare inter-cell channels

[n.cells]FOO state:  -- 'visible' cell state
#PRAGMA SHARED state -- allow parallel sharing
BOOL ok:             -- termination flag
#PRAGMA SHARED ok    -- allow parallel sharing

SEQ
  ...  initialise 'state' and 'ok'
  BARRIER b:
  PAR BARRIER b
    PAR i = 0 FOR n.cells BARRIER b
      cell (b, state[i], ok, ...)
    display (b, state, ok, ...)
```

The code outline for the individual '`cell`' processes is:

```
PROC cell (BARRIER b, FOO s, BOOL ok, ...)
  ...  declare and initialise local state
  WHILE ok   -- phase 0
    SEQ
      ...  interact with neighbours
      ...  and update local state
      SYNC b
      -- phase 1
      ...  update 's' (from local state)
      SYNC b
:
```

The code outline for the '`display`' process is:

```
PROC display (BARRIER b, []FOO s, BOOL ok, ...)
  ...  declare and initialise local state
  WHILE ok   -- phase 0
    SEQ
      ...  render observed cell states 's'
      SYNC b
      -- phase 1
      ...  if time to stop, set 'ok' false
      SYNC b
:
```

Reading the 'ok' flag by the '`cell`' and '`display`' processes happens in phase 0. In phase 1, the '`display`' process may decide to set the 'ok' flag to false. If that happens, all processes will see the modified flag in the same cycle and *gracefully* terminate together.

Rendering of the visible cell states, by '`display`', also happens in phase 0. Update of the visible cell states, by the '`cell`' processes happens in phase 1.

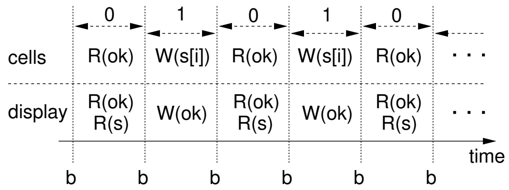

Fig. 3. Timeline of shared data access

Figure 3 shows the timeline of read and write operations performed by the various processes. This shows that access to the shared data adheres to the CREW rules.

Only two phases are used in this example — more complex systems may require more phases and coordination by more than one barrier. Providing that each phase, individually, has a parallel access pattern to shared data that sticks to CREW rules, the system has no race hazards. Such analysis (or, preferably, design) has *O(n)* complexity, where n is the number of *different* processes, and scales to very large and complex systems.

B. Language binding

The mechanism for synchronising access to shared data described above suffers from two difficulties:

- *the compiler* does not police the phased CREW access to shared data, making correct usage the responsibility of the programmer;
- *implementations* must be careful about local caching of the shared data.

On the first point, for correct policing of the CREW rules for each phase, the compiler would first need to detect that this concurrency paradigm was being used and, so, *needed* policing. A special language binding would help. Subsequent analysis might be eased by requiring the use of *different* barriers between each phase transitions (which has no run-time cost). Compiled usage information for processes could be extended to include read/write descriptions for the various items of shared data and the phases to which those descriptions apply. For separately compiled processes, this information would be included in the compiler output.

The second point requires a little more consideration. occam-π processes that have write access to data (e.g. the shared '`ok`' parameter for the '`cell`' process), normally expect that data to be *exclusive* to them — i.e. that the CREW rules are honoured at *process level* granularity. It is possible that on some architectures, the value of '`ok`' would be read once and stored in a processor register, with all subsequent reads (and writes in the display process) only affecting the register.

Fortunately, current implementations of occam-π (even with optimisations set at maximum levels) guarantee that this will not happen. No state is retained in processor registers across descheduling points, which now includes both `SYNC` and `RESIGN`. However, this may not necessarily be the case in the future.

Two different ways of handling this have been considered. Firstly, the compiler could generate hints about *volatile* data, allowing such to be flushed explicitly before a `SYNC` or `RESIGN`. This may become messy and complicates code generation. A second solution, which we are more likely to adopt, involves changing the nature of the shared data, such that reads and writes are forced to complete with respect to `SYNC` and `RESIGN` operations. This type of behaviour is already present in occam-π — *port* input and output, normally used for low-level hardware access. The existing language and compiler support for `PORT`s could be extended slightly to allow:

```
[n.cells]PORT FOO state:  -- 'visible' cell
PORT BOOL ok:             -- termination flag
```

These `PORT`s are meant to be shared — so no compiler directives are needed to say this. The '`state`' and '`ok`' ports would need slightly modified '`cell`' and '`display`' processes:

```
PROC cell (BARRIER b,
           PORT FOO s!, PORT BOOL ok?, ...)
  ...  declare and initialise local state
  BOOL running:
  SEQ
    ok ? running
    WHILE running   -- phase 0
      SEQ
        ...  interact with neighbours
        ...  and update local state
        SYNC b
        -- phase 1
        ...  "s ! from.local.state"
        SYNC b
        -- phase 0
        ok ? running
:
```

Note that the `PORT` parameters also include direction specifiers (`!`, `?`), explicitly declaring whether the shared data is read or written (or both) by a process.

The implementation of `PORT` inputs and outputs is trivial — simply memory reads and writes (with no synchronisation or locking required). CREW rules still must be applied and their safe operation is controlled by the barriers. Explicitly tagging the shared data as `PORT`s ensures that future code-generators will not cache their values in registers beyond another input or output from the same `PORT`. The extra cost of such an implementation would be minimal. Local variables, such as '`running`', could be cached in registers safely.

V. Performance, Conclusions and Future Work

Figure 4 shows the results of a benchmark that measures the time per barrier `SYNC` for increasing numbers of concurrent processes, run on 3.2 GHz. Pentium IV machines. Each process synchronises a fixed number of times, from which the average individual synchronisation time is calculated. The *stride* is used to control the start-up (and subsequent scheduling) order of parallel sub-processes, demonstrating the effect of the processor's cache pre-fetching.

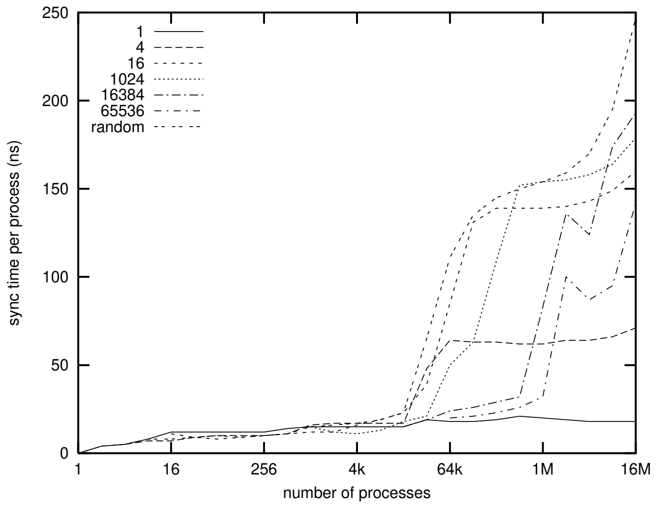

Fig. 4. Synchronisation time for different strides

The memory foot-print for the 16 million process benchmark (actually 2^{24}) was just over 700 mega-bytes (approximately 44 bytes per process), so cache-misses will be heavy. The processes are allocated their workspaces contiguously according to their index. The *stride* forces their scheduling so that their workspaces are (44**stride*) bytes apart. For small strides, the Pentium IV cache pre-fetching eliminates the problem of cache miss. For larger *stride*s, and especially for the *randomised* striding, the pre-fetching is defeated and cache miss penalties are felt.

Despite this, Figure 4 shows the implementation to be ultra-lightweight. The time for a *sixteen-million-wide* barrier synchronisation was only 16 ns per process in the best case (163 ms for the whole barrier) and 247 ns per process in the worst case. Typical application mixes will show *some* coherence in memory useage — the worst case above is really cruel! Also, applications running *real* processes (with real work to do) will not be able to afford more than the order of a million of them (because of memory limitations with current technology).

The barrier mechanisms presented in this paper are useful and fast. An important area for future research concerns the automated checking of shared variables whose access is made safe using barriers. Having the compiler guarantee the integrity of the design would relieve the designer of a certain amount of stress.

CSP allows any *event* to be used as a guard in a *choice*, including those bound to multiple parallel processes. Direct implementation in occam-π requires `SYNC` guards for use in `ALT`s. We know how to do this ([15], [16]), but will need to take care that the required protocols for correct execution are only set up when needed. Those protocols are not especially heavy — *except when compared* with our implementation of *committed* (i.e. non-`ALT`ing) barriers.

The current implementation does not respect process priority. It is a design constraint that processes synchronising on a barrier must all have the same priority level. This is currently unchecked, but run-time checks could easily be added (costing around 2 nanoseconds). Multi-priority barriers are possible, but require more internal storage (for 32 process-queues instead of the current 1) and a larger constant cost for barrier completion.

The description of barriers given here are *static* — they may not be communicated or assigned. We will want to use barriers in more volatile applications (e.g. for synchronising dynamically constructed *mobile* processes in some dynamic space-matrix [3]). We are currently investigating ideas for `MOBILE BARRIER`s that will give us this capability.

Future work also includes research into a *formal* CSP model for all barrier mechanisms in occam-pi. This will allow formal reasoning for the specification, refinement and verification of our systems, plus the use of the existing CSP model checker (FDR[18]).

References

[1] P. Welch and D. Wood, "The Kent Retargetable occam Compiler," in *Proceedings of WoTUG 19*. IOS Press, Mar. 1996, pp. 143–166, ISBN: 90-5199-261-0.

[2] P. Welch, J. Moores, F. Barnes, and D. Wood, "The KRoC Home Page," 2000, available at: http://www.cs.kent.ac.uk/projects/ofa/kroc/.

[3] P. Welch and F. Barnes, "Communicating mobile processes: introducing occam-pi," in *25 Years of CSP*, ser. Lecture Notes in Computer Science, A. Abdallah, C. Jones, and J. Sanders, Eds., vol. 3525. Springer Verlag, Apr. 2005, pp. 175–210, to appear.

[4] F. R. Barnes, "Dynamics and Pragmatics for High Performance Concurrency," Ph.D. dissertation, University of Kent, June 2003.

[5] F. Barnes and P. Welch, "Prioritised dynamic communicating and mobile processes," *IEE Proceedings – Software*, vol. 150, no. 2, pp. 121–136, Apr. 2003.

[6] Inmos Limited, "occam 2.1 Reference Manual," Inmos Limited, Tech. Rep., May 1995, available at: http://wotug.org/occam/.

[7] R. Milner, *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999, ISBN-10: 0521658691, ISBN-13: 9780521658690.

[8] C. Hoare, *Communicating Sequential Processes*. London: Prentice-Hall, 1985, ISBN: 0-13-153271-5.

[9] A. Roscoe, *The Theory and Practice of Concurrency*. Prentice Hall, 1997, ISBN: 0-13-674409-5.

[10] L. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[11] M. Schweigler, "Adding Mobility to Networked Channel-Types," in *Proceedings of Communicating Process Architectures 2004*, Sept. 2004, pp. 107–126, ISBN: 1-58603-458-8.

[12] P. H. Welch and D. C. Wood, "Higher Levels of Process Synchronisation," in *Proceedings of WoTUG 20*. IOS Press, Apr. 1997, pp. 104–129, ISBN: 90-5199-336-6.

[13] D. Wood and J. Moores, "User-Defined Data Types and Operators in occam," in *Proceedings of WoTUG 22*. IOS Press, April 1999, pp. 121–146, ISBN: 90-5199-480-X.

[14] M. Poole, "Extended Transputer Code - a Target-Independent Representation of Parallel Programs," in *Proceedings of WoTUG 21*. IOS Press, Apr. 1998, pp. 187–198, ISBN: 90-5199-391-9.

[15] P. Welch, "`ALT`ing on a `barrier` synchronisation," Private communication, Mar. 2003.

[16] J. Woodcock and A. McEwan, "On choice and multiway synchronisation," Private communication, Mar. 2004.

[17] B. Vinter, O. J. Anshus, and T. Larsen, "Pastset: A distributed structured shared memory system," in *Proceedings of High Performance Computers and Networking Europe*, Amsterdam, The Netherlands, Apr. 1999.

[18] *FDR2 User Manual*, Formal Systems (Europe) Ltd., 3, Alfred Street, Oxford. OX1 4EH, UK., May 2000.