Communicating Process Architectures – 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

Prioritised Dynamic Communicating Processes: Part I

Fred BARNES and Peter WELCH
Computing Laboratory, University of Kent, Canterbury, KENT. CT2 7NF
{frmb2,phw}@ukc.ac.uk

Abstract. This paper reports continuing research on language design, compilation and kernel support for highly dynamic concurrent reactive systems. The work extends the occam multiprocessing language, which is both sufficiently small to allow for easy experimentation and sufficiently powerful to yield results that are *directly* applicable to a wide range of industrial and commercial practice. Classical occam was designed for embedded systems and enforced a number of constraints – such as statically pre-determined memory allocation and concurrency limits – that were relevant to that generation of application and hardware technology. Most of these constraints have been removed in this work and a number of new facilities introduced (channel structures, mobile channels, channel ends, dynamic process creation, extended rendezvous and process priorities) that significantly broaden occam's field of application and raise the level of concurrent system design directly supported. Four principles were set for modifications/enhancements of the language. They must be useful and easy to use. They must be semantically sound and policed (ideally, at compile-time) to prevent mis-use. They must have very lightweight and fast implementation. Finally, they must be aligned with the concurrency model of the original core language, must not damage its security and must not add (significantly) to the ultra-low overheads. These principles have all been observed. All these enhancements are available in the latest release (1.3.3) of KRoC, freely available (GPL/open source) from: www.cs.ukc.ac.uk/projects/ofa/kroc/.

1 Preamble

This paper is the first of two describing the various dynamic and priority enhancements to occam. This paper concentrates on the extensions themselves, whilst the second paper [1] gives examples of how they are used. Also included in the second paper is a quick overview of other (less significant) additions and extensions to the language and compiler.

2 Introduction and Motivation

This paper describes five extensions to the occam [1]/CSP [2, 3, 4, 5] programming language: process priority, mobile channel-types, channel direction specifiers, dynamic process creation and an extended rendezvous. The dynamic memory mechanisms also allow the introduction of *recursive* processes and run-time sized `PAR` process replication. The latter require no significant language change and are described in [1] – along with the tidying-up of a number of additional items from the original occam language.

The occam [4] programming language, based on the CSP [2, 3] process algebra, encourages programmers to build applications as layered networks of active/reactive components

[1]Trademark of ST Microelectronics

(processes), synchronising and communicating through channels. occam channels are point-to-point, synchronised, unbuffered communication links. Figure 1 shows an example component, a *running-sum integrator*, implemented as a simple network of three sub-processes, 'plus', 'delta' and 'prefix(0)'. The external interface is provided through two channels 'in' and 'out'.

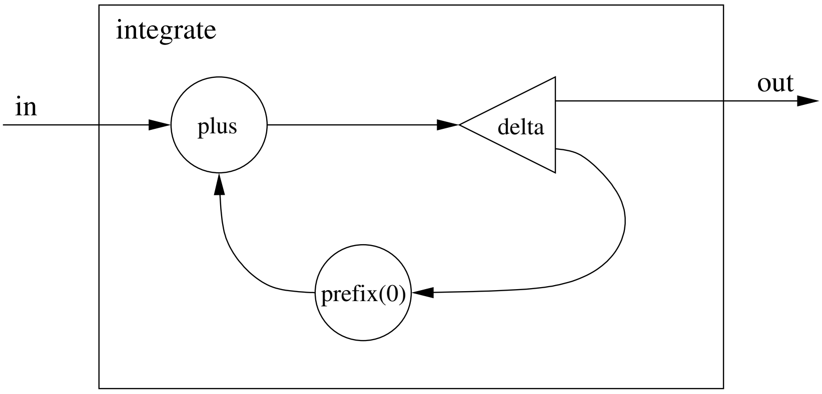

Figure 1: occam integrate process network

The traditional method of constructing occam programs follows this simple design, i.e. *process orientated*. Process orientated in this context is the construction of programs as networks of independent active processes, that communicate and/or synchronise using the primitives available. In the KRoC [6] implementation of occam, these synchronisation primitives are primarily channel communications and implicit barriers (on PAR constructs), plus additional non-compiler based synchronisations (semaphores, barriers, buckets and resources), as described in [7].

While these capabilities are sufficient for the design and implementation of many occam programs, certain types of application remain hard – most notably applications where dynamic allocation is desirable. Plain occam also lacks any concept of a pointer-type, which has memory and run-time implications for communication, where data must be copied[2]. The introduction of *mobile* data-types to occam [8, 9], that provide safe non-aliasing *movement* semantics implemented using pointer manipulation, as well as the ability to create run-time sized mobile arrays, helps significantly.

Dynamically reconfiguring a process network is still *hard* however. A (statically) limited form of dynamic process creation has always been available, in terms of choices and PAR. So, the 'integrate' process shown in figure 1 may possibly replace its 'plus', 'prefix' and 'delta' network with something else. Such decisions remain internal to 'integrate' and are not the concern of anything connected to the 'in' or 'out' channels.

Channel-types (first proposed for the un-implemented occam 3 [10] language) provide a method of grouping together a number of related channels within a single type – for example, channels which define a client/server interaction. Our channel-types extend this idea – however, channel-type variables reference only one of their *ends* and those *ends* are mobile. To distinguish between the two different ends, a *channel direction specifier* ('?' or '!') is added to the type when declaring the channel-end. These channel-ends are similar in idea to Icarus' *ports* [11, 12], except that we allow an arbitrary number of channels within a single type. The channel-types added to occam are treated as first-class citizens in the type system (like any

[2]It is possible to avoid copying on communication (and assignment) by transparently providing managed copy-on-write semantics, along with some form of reference-counted garbage collection. This has not been examined in detail yet however.

mobile variable), allowing channel-ends of that type to be declared and subsequently communicated to other processes. Assignment and parameter-passing are handled in a similar (*mobile*) way. Channel-ends (either side) may also be `SHARED` (another occam 3 idea) by many processes, with security enforced by the language design.

Section 4 describes the syntax, semantics and implementation of channel-types in occam, within the framework of KRoC/Linux [13].

Simple channel communication in occam is described in CSP by participation in an event (the channel). The outputting process readies the data then synchronises on the channel. The inputting process synchronises on the channel then uses the data. The point at which both processes synchronise is viewed as an instantaneous action where the data (or rights to the data) are copied. After the data has been transferred, both the inputting and outputting processes resume execution.

Sometimes it is desirable to have the inputting process perform some action on the received data, before the outputting process resumes execution. Currently, the only way to do this is to alter the inputting and outputting processes so that they perform an additional synchronisation at the required point. However, doing this is intrusive to the outputting process (which doesn't need to know such details about the inputting process), and there is every possibility of a small misplacement of the second synchronisation on either side causing deadlock.

The *extended rendezvous* offers a nice solution to this problem, since it requires no adjustment of any outputting process and provides a safe (compiler-checked) syntax and semantics for the inputting process. Section 5 describes this in detail.

Standard occam only allows the creation of new processes inside a `PAR` construct, which forces them to synchronise at the end of the `PAR` on a barrier. This also applies to the replicated `PAR`, whose replicator count must be a compiler-known constant expression. The follow-on paper [1] describes an extension for handling run-time-count replicated `PAR`s, a description of which can also be found in [14]. This greatly enhances the expressiveness of a replicated `PAR`, but also makes it more difficult for the compiler to perform static checks on parallel usage within the replicated process.

We introduce a new mechanism for process creation, the `FORK`, which dynamically creates a `PROC` instance and runs it parallel to the invoking process. Locally, `FORK` behaves as *Skip* (a do-nothing process), with some minor exceptions. Termination of a `FORK`ed process is controlled by a `FORKING` block outside any `FORK`s. When the `FORKING` finishes, it waits for any unfinished `FORK`ed processes. A `FORK`ed process may finish early however, allowing dynamically allocated resources to be re-used immediately.

Section 6 describes the syntax, semantics and implementation of the `FORK`. A typical application for `FORK`s is in internet servers, where it is highly desirable to be able to spawn a new process for handling an incoming connection. An experimental version of the occam web-server [15] is under construction, using `FORK`s and channel-types.

With the introduction of dynamic channels and process-creation, the use of priority becomes much more prevalent. For example, we might want the occam web-server to give priority to connections from academic institutions before connections from elsewhere. This could easily be done by having the connection-handling processes dynamically change their own priority when handing a (locally) new connection. This version of priority has been added into the language through the use of compiler built-ins (plus significant changes in the run-time system). This form of priority was first investigated in [16]. Section 7 describes our implementation of priority.

3 Channel Direction Specifiers

As mentioned previously, channel direction specifiers are used in the declaration of channel-type variables to distinguish the *client* ('!') or *server* ('?') nature of that endpoint. Another use of these specifiers is on regular channel parameters, to indicate the direction of data-flow. Channel direction specifiers used in this way were first discussed in [17]. Normally, occam channel parameters, as they appear in `PROC` headers, don't specify the direction of the channel, even though only one end-point of that channel is really there. The compiler performs extensive checks to 'work out' the usage of channels, needed so that it can check any parallel misuse of a particular channel. From the compiler point-of-view, direction specifiers don't actually add anything useful. Their use is intended for the user and to improve the clarity of occam programs. For example, the `PROC` header for '`integrate`' in figure 1 would read (assuming integer input and output):

```
PROC integrate (CHAN INT in, out)
```

From the programmer's point-of-view, there is probably enough information here to readily use the component – the channels are obviously named. However for less obvious named parameters (e.g. '`data`' and '`count`'), it is not apparent which way data is communicated. To avoid such ambiguities in reality, parameters are often renamed such that their direction is incorporated, (as we have already done for the '`integrate`' process). However this does not prevent someone accidently mis-naming a parameter.

Channel direction specifiers add to the 'type' of the channel, not to its name, and the compiler can check this against how the parameter is actually used, both inside '`integrate`' and in any external instances of '`integrate`'. The direction is specified by placing the input ('?') or output ('!') operator against the name. The modified '`integrate`' `PROC` header would read:

```
PROC integrate (CHAN INT in?, out!)
```

From this modified specification of the process, it is immediately obvious in which direction the '`in`' and '`out`' channels are used, even in the case of a less obvious parameter naming scheme. Additionally, channel direction specifiers bring the `PROC` header closer to the diagram which represents it (figure 1), which is good from a software engineering point of view.

Channel direction specifiers are also used when referring to channel variables in parameters (and abbreviations). This adds a wealth of information to the implementation of a process. For example, the original '`integrate`' implementation may have been:

```
PROC integrate (CHAN INT in, out)
  CHAN INT a, b, c:
  PAR
    plus (in, c, a)
    delta (a, out, b)
    prefix (0, b, c)
:
```

Although parameters are sensibly named, channel variables are often not – related to the fact that a channel declaration declares two conceptual *ends*, but both with the same name. If asked to draw a picture of the above network, one would need to look at the `PROC` headers for the '`plus`', '`prefix`' and '`delta`' processes.

As mentioned previously, channel parameters and abbreviations can only ever refer to one particular end. Channel direction specifiers put this channel usage information into the program, bringing the code closer to the design.

The above code fragment, with channel direction specifiers, would read:

```
PROC integrate (CHAN INT in?, out!)
  CHAN INT a, b, c:
  PAR
    plus (in?, c?, a!)
    delta (a?, out!, b!)
    prefix (0, b?, c!)
:
```

Figure 1 could now be drawn directly from this above code – no reference to the definition of the sub-components need be made. If there were a mismatch, the above would not compile.

Channel direction specifiers are designed to help the programmer. The current implementation performs additional checking where channel direction specifiers are present, ensuring that everything is consistent. Their usage is entirely optional, but can be enforced by running the compiler in *strict* mode, which requires that channel direction specifiers are present wherever possible. There is one other gain with channel direction specifiers, which is their use in externally defined `PROC`s (used to interface external code, e.g. C, with occam [18]). Passing channels outside the occam world is dangerous at best, but at least with channel direction specifiers we can make sure that the right ends are used.

3.1 Direction Specifying Channel Arrays

Specifying the channel direction in a channel-array parameter is no problem, for example:

```
PROC merge ([]CHAN BYTE in?, out!)
```

All elements of a channel array must be used in the same direction.

However, specifying the direction of array subscriptions and slices presents a small syntax issue, of whether the direction should be specified with:

```
SEQ
  out.string ("hello ", 0, out[0]!)
  multi.out.string ("world!*n", 0, [out FROM 1 FOR 2]!)
```

or with:

```
SEQ
  out.string ("hello ", 0, out![0])
  multi.out.string ("world!*n", 0, [out! FROM 1 FOR 2])
```

We have chosen a mixed implementation, which we believe to be the most obvious. Channel array subscriptions use the former syntax, slices use the latter syntax. Although the latter syntax of "`out![0]`" would be more consistent with subscriptions, it does look a little peculiar (more like an output than an array subscription!). In addition, it turns out that parsing subscriptions in an "`out![0]`" form is particularly tricky, as the compiler will try and parse it as an output. It is possible to change the parser to cope with this, but requires moving some of the checks (including a syntax check) into the type-checker once the type of '`out`' has been determined. This may not be entirely desirable.

It has been pointed out that using the syntax of "`out[0]!`" for channel-array subscriptions suggests we are referring to the output-end of the channel '`out[0]`' as opposed to index 0 of the output-only channel array '`out!`'. This is not the case however – the latter meaning applies. In fact, the former meaning has no syntax in the language. Once declared, any subsequent use of a channel must refer to a single-end only. Previously this has always been implicit and something deduced by the compiler. This limitation goes both ways, as such an array of channel-ends is exclusively always input or output, mixtures are not allowed.

Abbreviations follow the style of parameters (they are semantically the same thing), for example:

```
[4]CHAN INT chans:
SEQ

  []CHAN INT if.chans? IS [chans? FROM 1 FOR 3]:
  CHAN INT ix.chan? IS chans[0]?:
  PAR
    ...
```

4 Mobile Channel-Types

Here is an example of a *channel-structure*:

```
CHAN TYPE BUF.MGR
  MOBILE RECORD
    CHAN INT req?:              -- carries integers (size of requested buffer)
    CHAN MOBILE []BYTE buf!:    -- carries dynamically sized arrays
    CHAN MOBILE []BYTE ret?:    -- carries dynamically sized arrays
:
```

This declares a *mobile* channel-type called 'BUF.MGR'. Being a channel-type, the fields inside the RECORD structure are only permitted to be channels (or arrays of channels). *Data* fields are not allowed.

'BUF.MGR' contains three channels. 'req' is used by a *client* to request a buffer of some size, which is acquired from the 'buf' channel. Once the client is done with the buffer, it sends it back to the *server* using the 'ret' channel. The channel direction specifiers specify *server*-relative directions. The server side can only use 'req' and 'ret' for input and 'buf' for output. In contrast, the client side can only use 'req' and 'ret' for output and 'buf' for input. This behaviour is enforced by the compiler, as is the placement of direction specifiers on the fields within the channel-structure.

We use the words '*client*' and '*server*' here to mean a particular *end* of the channel-type. Whether the application chooses to use them as such is entirely up to the application. We would prefer, however, that they were used according to a well-understood usage pattern, such as client-server, IO-SEQ or IO-PAR [19].

4.1 Variables and Allocation

Channel-type variables come in two forms – a *server-end* and a *client-end*. For example:

```
BUF.MGR? buf.svr:               -- server-end variable
BUF.MGR! buf.cli:               -- client-end variable
```

A direction specifier is used on the *type* to indicate either a server ('buf.svr') or a client variable ('buf.cli'). Once declared, they are *undefined* until either allocated or used as a target of assignment or input – remember that these are *mobile* variables and have the same underlying semantics as the mobile *data-types* described in [9]. They are allocated in pairs at run-time:

```
buf.cli, buf.svr := MOBILE BUF.MGR
```

This operation dynamically allocates the channel-structure record and assigns it to both target variables. Their usage in assignment and communication is strictly controlled by the rules for ordinary `MOBILE` variables – i.e. a movement semantics. Special care needs to be taken when handing the freeing of these dynamic mobiles – this is discussed in section 4.4. The compiler is not fussy about the order in which the client and server variables appear on the left-hand side, but does check to ensure that one is a client and the other is a server. This allocation syntax is similar to general dynamic mobile allocation [9], except that in this case there is no array dimension to be specified.

The variables '`buf.svr`' and '`buf.cli`' can be used in two ways. Either as themselves in communication, assignment, parameter passing and abbreviations; or in a subscripted expression to access individual channel fields. The two are mutually exclusive. Firstly because any attempt at a combination would fail in alias analysis and secondly because we do not allow recursive channel types (e.g. '`BUF.MGR`' having a '`CHAN OF BUF.MGR!`' field).

4.2 Using Channel-Types

For the most part, channel-type variables can be treated like ordinary mobile variables. Once allocated, this means that moving one of the channel-type ends around the network *stretches* the channels within it. The behaviour is conceptually like the similar mechanism in Icarus [11, 20], but our implementation differs significantly.

Figure 2 shows a simple occam program which allocates a mobile channel-type, then communicates its ends to other processes, which use the channels contained within those ends to communicate directly with each other. It uses the same `BUF.MGR` type described at the start of section 4. All `PROC` parameters and channels passed to them have been augmented with channel-direction specifiers.

Figure 3 shows the state of things after the '`generator`' process has allocated the channels, but before it has communicated them. The '`generator`' process then moves the server-end to the '`server.process`' process. This has the effect of pulling the channels 'over' the network, as shown in figure 4. The client-end is then moved to the '`client.process`' process and the '`generator`' process terminates. The two processes ('`server.process`' and '`client.process`') then proceed to communicate over the channels connecting them. This final network configuration is shown in figure 5.

This example is not particularly exciting, but demonstrates how channel-types can be used. The main point about this is that we don't necessarily need to know where the remote end of a channel-type is. Indeed, one of the general points of occam/CSP is that we should *not need* to know where channels are connected, only what protocols and usage patterns they have[3].

4.3 Shared Channel-Types

The channel-types presented so far provide a general mechanism for moving channels around networks, neatly grouped according to function. A common thing we tend to do with channels is to share them. Sharing of channels (and any other variable, parameter or abbreviation) is currently performed using a compiler directive to turn off usage checking and a `SEMAPHORE` (user-defined) [22] type to provide mutual exclusion. We would clearly wish to avoid this approach, since it removes any opportunity for the compiler to check affected code for aliasing and parallel usage.

[3]However, we note that there are dangers in allowing network topologies to be set up dynamically [21] – particularly in regards to deadlock/livelock analysis.

```
PROC server.process (CHAN BUF.MGR? in?)
  BUF.MGR? sv:
  SEQ
    in ? sv                             -- get server-end channels

    INT s:
    MOBILE []BYTE b:
    SEQ
      sv[req] ? s                       -- get size
      b := MOBILE [s]BYTE               -- allocate buffer
      sv[buf] ! b                       -- send buffer to client
      sv[ret] ? b                       -- take buffer back
:

PROC client.process (CHAN BUF.MGR! in?)
  BUF.MGR! cv:
  SEQ
    in ? cv                             -- get client-end channels

    MOBILE []BYTE b:
    SEQ
      cv[req] ! 1518                    -- send desired buffer size
      cv[buf] ? b                       -- get buffer
      ...  use ‘b’
      cv[ret] ! b                       -- send buffer back
:

PROC generator (CHAN BUF.MGR? s.out!, CHAN BUF.MGR! c.out!)
  BUF.MGR? buf.svr:
  BUF.MGR! buf.cli:
  SEQ

    buf.cli, buf.svr := MOBILE BUF.MGR   -- create channels

    s.out ! buf.svr                     -- send server channels
    c.out ! buf.cli                     -- send client channels
:


CHAN BUF.MGR? svr.chan:
CHAN BUF.MGR! cli.chan:
PAR
  generator (svr.chan!, cli.chan!)
  server.process (svr.chan?)
  client.process (cli.chan?)
```

Figure 2: Simple mobile channel-type demonstration program

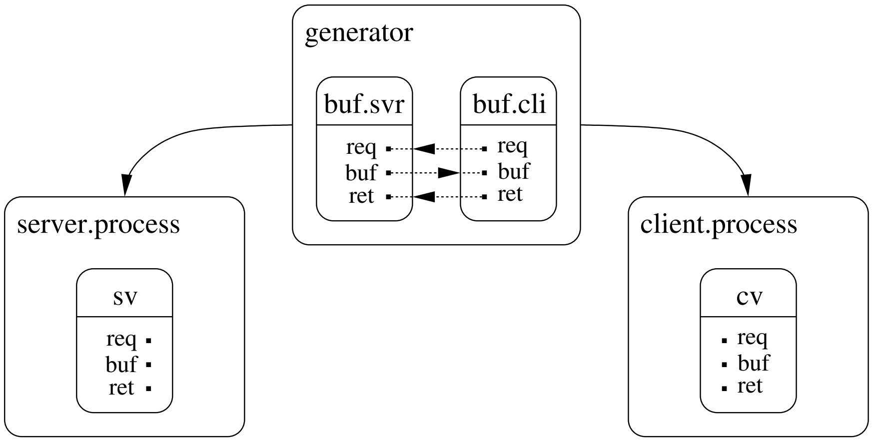

Figure 3: Process states after channel allocation

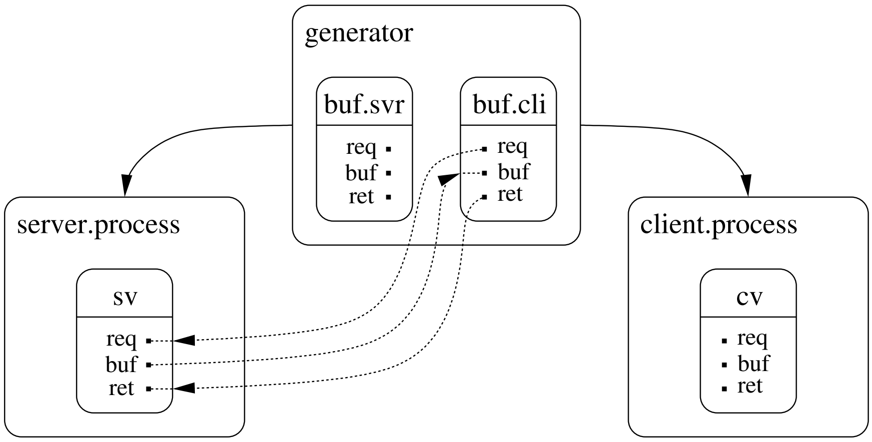

Figure 4: Process states after moving the server-end

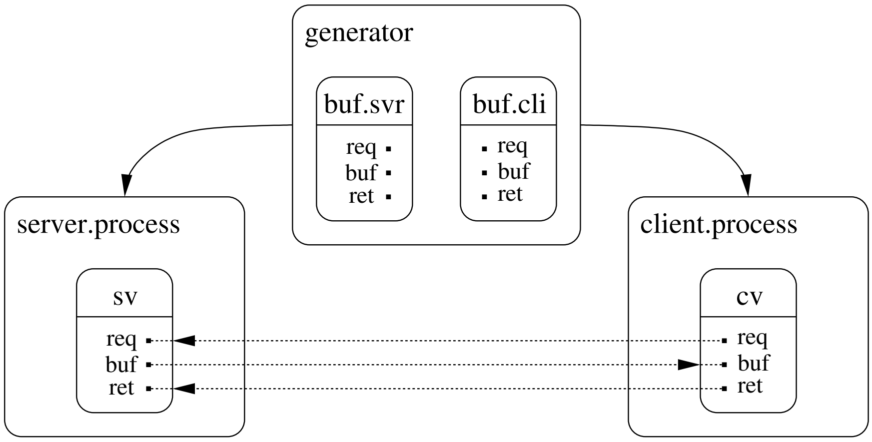

Figure 5: Final process states after moving the client-end

We solve this problem (for channel-types at least), by allowing the declaration of SHARED channel-type variables (of a channel-structure) and subsequently enforcing their safe use. Either the client or server ends may be shared, or both, providing the full set of *one-to-one*, *any-to-one*, *one-to-any* and *any-to-any* channel arrangements (similar to those in JCSP [23]). For example, an *any-to-one* channel-type pair can be created with:

```
SHARED BUF.MGR! s.cli:
BUF.MGR? u.svr:
SEQ
  s.cli, u.svr := MOBILE BUF.MGR
```

The 'SHARED' attribute changes the nature of the type, such that it is only compatible with other *shared* channel-types of the same CHAN TYPE and *endianism*. This prevents the accidental mix-up of shared and non-shared channel-types, which would be disastrous.

Before the channels inside a *shared* channel-structure may be used, the whole channel-structure (or rather the relevant channel-type *end* of it), must be CLAIMed. This follows a style similar to that presented in occam3 [10], although what we are CLAIMing is a little different. For example, a new 'client.process' which operates on client-shared channel-types might look like:

```
PROC client.process.2 (SHARED BUF.MGR! cv)
  MOBILE []BYTE b:

  CLAIM cv                      -- claim it
    SEQ
      cv[req] ! 1518            -- send desired buffer size
      cv[buf] ? b               -- get buffer
      ...  use 'b'
      cv[ret] ! b               -- send buffer back
:
```

Whilst this process is in the body of the CLAIM, other clients are blocked from using the channel-structure ('cv'). The same applies to shared *server-ends* too.

The usage rules for CLAIM differ slightly depending on whether the CLAIMed channel-structure is a *server* or *client* end. Once it has claimed the client end of a channel-type, a process may only communicate on the channels within *a* CLAIMed structure and must not CLAIM anything else. Assignment, function-calls and timeouts are still permitted, as are PROC calls (on the condition that any channels used are part of the CLAIMed structure).

The rules for the server CLAIM are slightly different. Once a process CLAIMs a server-shared end it must not make any nested CLAIMs on other (shared) server ends. It may however CLAIM client-shared ends and act as a client to other servers. Usage of other channels within the body of a server CLAIM is unrestricted. The issue of cyclic deadlock exists here, when a loop of client-server relationships form, but this can be avoided by careful design [19].

In the case of long-running transactions, an *any-to-any* channel becomes less useful, since both the client and server must remain in the CLAIM for the duration of the transaction, preventing other clients and servers interacting. However, what we can do is create another *any-to-any* channel-type, which only communicates the *one-to-one* chan-types for each client/server pair to use privately. For example:

```
CHAN TYPE C.BUF.MGR
  MOBILE RECORD
    CHAN BUF.MGR? svr?:
:
```

This allows an *any-to-any* channel carrying `BUF.MGR?`s to be created. When a client process wishes to engage in a transaction with a server process, it can dynamically create the `BUF.MGR` channels, communicate the server-end and use the client-end locally. The '`client.process`' process in this case could be written as:

```
PROC client.process.3 (SHARED C.BUF.MGR! c.cv)
  BUF.MGR! cv:
  BUF.MGR? sv:
  SEQ
    cv, sv := MOBILE BUF.MGR     -- allocate channels

    CLAIM c.cv                   -- claim connection to (any) server
      c.cv[svr] ! sv             -- send it the server end

    MOBILE []BYTE b:
    SEQ
      cv[req] ! 1518             -- send desired buffer size
      cv[buf] ? b                -- get buffer
      ...  use 'b'
      cv[ret] ! b                -- send buffer back
:
```

This also removes the need for explicitly shared `BUF.MGR`'s, since a new set of channels is created for each transaction. Of course, the clients and servers are free to re-use existing channels as they see fit. Conceptually this is like adding "another level" of channels to the network, which can be moved around by means of channel communication on the level below.

To create an alias of a (shared) channel-end, the mobile '`CLONE`' operator is used. This may seem counter-intuitive, since `CLONE` is used on mobile data types to create an actual copy. There is no sense in making a copy of a channel-type though – it doesn't contain any data to copy. Because of this, we recycle the `CLONE` operator to create aliases for shared channel-types. The use of `CLONE` is still restricted in the same way it is for mobile data-types, with the exception that we allow its use in parameters and abbreviations.

The corresponding (possibly shared) server process for '`client.process.3`' simply inputs the server-end manufactured in the client then uses that for communication:

```
PROC server.process.3 (SHARED C.BUF.MGR? c.sv)
  BUF.MGR? sv:
  SEQ
    CLAIM c.sv                   -- claim connection from (any) client
      c.sv[svr] ? sv             -- input server end

    MOBILE []BYTE b:
    INT s:
    SEQ
      sv[req] ? s                -- input size
      b := MOBILE [s]BYTE        -- allocate buffer
      sv[buf] ! b                -- send to client
      sv[ret] ? b                -- take back when client done
:
```

The network connecting these together looks like:

```
SHARED C.BUF.MGR! cli.c:
SHARED C.BUF.MGR? svr.c:
SEQ

  cli.c, svr.c := MOBILE C.BUF.MGR       -- allocate channel

  PAR i = 0 FOR 4                        -- 4 clients and 4 servers
    PAR
      client.process.3 (CLONE cli.c)
      server.process.3 (CLONE svr.c)
```

This example creates a network in which four client processes and four server processes are plugged into a shared channel-structure (of type '`C.BUF.MGR`'). When the client wishes to interact with *any* server, it simply allocates the (unshared) '`BUF.MGR`' channel required for communication and communicates the server end through the shared '`C.BUF.MGR`' channel-type.

4.4 Memory Management

Mobile channel-types are implemented using pointers, in a similar way to mobile data-types. To control the aliasing and freeing of allocated memory, a reference-count [24, 25] based solution is used. Channel-type variables are pointers in the process workspace, initialised to *null*. The special allocation operation, "`cv, sv := MOBILE BUF.MGR`", allocates and initialises a block of memory, storing the resulting pointer in both these variables. Figure 6 shows the layout of the `BUF.MGR` block after it has been allocated in an *any-to-one* configuration (shared clients).

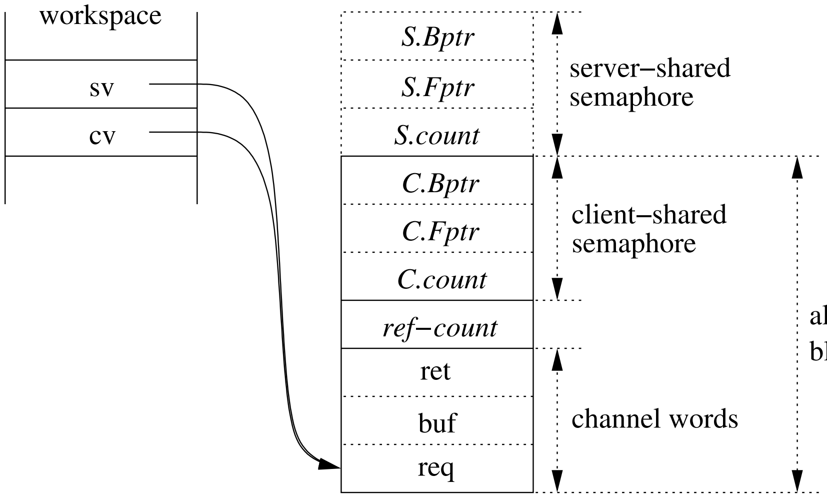

Figure 6: Layout of the `BUF.MGR` channel-type with a shared-client and non-shared server.

Immediately after allocation, '*ref-count*' is set to 2 (the two local references) and the client semaphore is initialised. If the channel-type is server-shared the server semaphore would be initialised here too. '*ref-count*' is incremented each time one of the pointer variables is `CLONE`ed and decremented when the variables leaves scope, or when it is used as a target of assignment or input. When the reference-count reaches zero the memory is freed.

Memory is allocated and freed using the existing Brinch-Hansen style allocator [26, 27], grouped by half-powers of 2. This is accessed using the existing 'MALLOC' and 'MRELEASE' instructions [9].

The cost of communication on one of these channels is only slightly more than the cost of a regular communication. The extra cost is incurred in loading the channel pointer, which will usually just involve adding a constant offset to the loaded pointer.

4.5 Non-Dynamic Non-Mobile Channel-Types

In the less dynamic environment, such as an embedded system, real dynamic memory may not be available. Having the benefits of channel-types is still desirable though. There are two solutions to this. The first is to restrict the size of the dynamic memory pool, making the 'MALLOC' instruction a possible descheduling point (although this is currently unimplemented). The second is to make the type non-mobile which causes it to exist in the local workspace of a process. The second option is examined here.

Making the channel-type non-mobile means that we can no longer *move* it around. Which rules out communication and assignment. Because of this, a single name can refer unambiguously to both ends of the channel. The type and variable declarations are:

```
CHAN TYPE FOO
  RECORD
    CHAN INT req?:
    CHAN BYTE resp!:
:

FOO c:
```

Because no direction is specified on the type of 'c', it must be added whenever 'c' is subsequently referred to (just as for normal channels). Channel-type parameters still carry the direction in the type since they can only ever refer to one end. For example:

```
PROC foo.svr (FOO? link)
  INT x:
  SEQ
    link[req] ? x
    ...
    out.string (.., 0, link[resp]!)
:

...

FOO c:
PAR
  foo.svr (c?)
  ...
```

When using a channel field directly for communication, no direction specifier is needed since the channel-direction is specified in the channel-structure (plus it doesn't fit into the language syntax for communication). The direction should still be specified when the channel is used as a parameter, however, as shown in the example above.

5 The Extended Rendezvous

The extended rendezvous is a mechanism for allowing the inputting process of a communication to execute a process with the communicated data, whilst the outputting process remains suspended. A new *extended-input* process implements this. It is syntactically similar to an ordinary input, but with '`??`' instead of '`?`'. However, it is followed by a compulsory indented process (the *extended-rendezvous*), which is executed whilst the outputting process remains blocked.

One application of this is that it allows us to *tap* a channel in a way that does not affect the synchronisation between the processes either side. This is useful when we wish to 'inspect' the data flowing round a process network – channels connecting existing processes can be 'tapped' without changing the semantics of that process network. (Assuming of course that the processes monitoring the *tapped* output channels guarantee always to take it). Figure 7 shows how we might inspect communication happening on the '`squares`' process pipeline of [28].

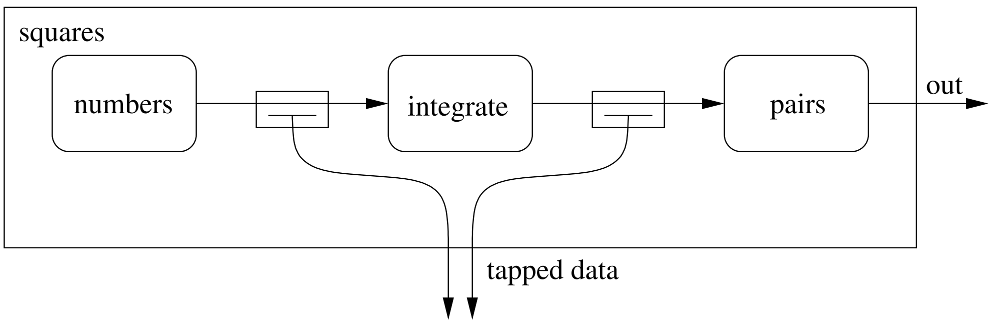

Figure 7: Tapped '`squares`' process pipeline

Here is the code for the '`tap`' process, assuming `INT` data-flow:

```
PROC tap (CHAN INT in?, out!, tap!)
  WHILE TRUE
    INT x:
    SEQ
      in ?? x
        out ! x
      tap ! x
:
```

5.1 Semantics

The semantics of the extended rendezvous are quite simple. Consider the following input and output processes running in parallel:

```
c ! 42                          c ?? b
                                  ...  extended rendezvous (no c?)
```

This has the same semantics as the following pair of processes:

```
SEQ                             SEQ
  c ! 42                          c ? b
  c.ack ? any                     ...  extended rendezvous (no c?)
                                  c.ack ! TRUE
```

where 'c.ack' is an extra CHAN BOOL on which the processes synchronise. The implementation is quite different, but the semantics remain as presented here.

5.2 ALTs and CASE Inputs

An extended rendezvous may be used as a guard in an ALT:

```
ALT
  c ?? x
    ...  extended rendezvous (no c?)
    ...  guarded process (optional)
  d ? y
    ...  guarded process
```

In the above, if the first guard is chosen, the outputting process is rescheduled after the extended rendezvous. The guarded process (at the same level of indentation as the extended rendezvous process) is then executed. Often, there is nothing left to do after the rendezvous process. Instead of writing SKIP, the guarded process may *in this case* be omitted.

A similar construction is needed when using variant (CASE) protocols:

```
PROTOCOL CONTROL
  CASE
    data; INT
    stop
:

...

in ?? CASE
  INT x:
  data; x
    ...  extended rendezvous (no in?)
    ...  case process (optional)
  stop
    ...  extended rendezvous (no in?)
    ...  case process (optional)
```

In both tag cases, the second indented process is executed after the outputting process has resumed. This second process is always optional and if not present is assumed to be *Skip*.

5.3 Usage Restrictions

There is only one usage restriction on extended-input: the channel undergoing the extended input may not be used in the *extended-process*. Any attempt to use the channel involved would immediately deadlock, since the process on the other end of the channel is suspended. This restriction is enforced by the compiler.

5.4 Implementation

The implementation of the extended rendezvous has required a number of additional *pseudo*-Transputer instructions, which are handled through appropriate entry-points in the run-time kernel. Table 1 gives a brief list of the instructions added. However, no change is required in the outputting process (needed for separate compilation) and there is *zero* impact on the cost of ordinary (non-extended) communication.

To make the implementation easy, we arrange for the extended input ('XIN') to always arrive second at the channel. This could be done by setting up a one-branched ALT construct. We shortcut this with a new special instruction ('XABLE').

Mnemonic	Parameters	Description
XABLE	*chan-addr*	enable
XIN	*count, chan-addr, dest-addr*	input
XMIN	*chan-addr, dest-addr*	mobile input
XMIN64	*chan-addr, dest-addr*	dynamic mobile input
XMINN	*count, chan-addr, dest-addr*	multi-dim dynamic mobile input
XEND	*chan-addr*	finish (resumes outputting process)

Table 1: Additional instructions to implement the extended rendezvous

For example, here is a simple extended rendezvous:

```
in ?? v
  P ()
```

This generates the following sequence of instructions:

```
LD in              -- load channel address
XABLE              -- extended enable

LD ADDRESSOF v     -- load destination address
LD in              -- load channel address
LD count           -- load (byte) count
XIN                -- extended input

CALL P             -- do P process

LD in              -- load channel address
XEND               -- resume outputting process
```

The actual code generated is likely to differ, depending on the nature of the 'in' and 'v'. The only descheduling instruction is 'XABLE'. This *waits* for the outputting process to arrive at the communication, or returns immediately if the outputting process is already in the channel. XABLE pretends to be an ALTing process when waiting, which causes an outputting process to reschedule the ALTer, rather than communicate (if it arrives second).

The 'XIN' instruction simply copies the data from the outputting process, or in the case of 'XMIN' *moves* the data from the outputting process. With the data in hand, P is run before calling 'XEND' which reschedules the blocked outputting process and clears the channel.

When it occurs in an ALT, the 'XABLE' call can be omitted – the outputting process *must* be in the channel word if the guard fired. ALTing will still incur the cost of the relevant 'ENBC''and DISC' handling, but this can be optionally inlined to minimize overheads.

Sequential and tagged protocols require slightly different code-generations, since we only really need to perform the extended input once (protocols are implemented as sequential communication). For sequential protocols, only the last input generates the extended sequence, preceding inputs are handled as per usual. For example, the sequential protocol and extended input:

```
PROTOCOL SEQP IS INT; MOBILE []BYTE:

...

INT x:
MOBILE []BYTE b:
in ?? x; b
  P (..)
```

generates code equivalent to (the illegal):

```
INT x:
MOBILE []BYTE b:
SEQ
  in ? x
  in ?? b
    P (..)
```

Case inputs follow a similar implementation, except for cases where the protocol contains a data-less tag. For these, the first input generated is also extended, since it may be the last input, but we won't know until the tag has been inspected (which arrives on the first communication). For example, the tagged (variant) protocol and extended input:

```
PROTOCOL VAR.PROTO
  CASE
    empty
    buffer; MOBILE []BYTE
:

...

in ?? CASE
  empty
    P.empty (..)          -- extended rendezvous (no in?)
    Q.empty (..)          -- case process (optional)
  MOBILE []BYTE b:
  buffer; b
    P.buffer (..)         -- extended rendezvous (no in?)
    Q.buffer (..)         -- case process (optional)
```

generates code equivalent to (the illegal):

```
BYTE $tag:                -- compiler generated name
in ?? $tag
  CASE $tag
    empty
      SEQ
        P.empty (..)      -- extended rendezvous
        ...  do XEND      -- reschedule outputting process
        Q.empty (..)      -- case process (optional)
    buffer
      MOBILE []BYTE b:
      SEQ
        ...  do XEND      -- reschedule outputting process
        in ?? b
          P.buffer (..)   -- extended rendezvous
          Q.buffer (..)   -- case process (optional)
```

Since 'XEND' is called additionally within each tag, it is not generated for the tag input itself. This has the hidden advantage that if a process outputs a data-less tag which the inputter doesn't handle, the outputting process will be left blocked in the channel (trying to communicate the tag, from its point of view). The inputting process will generate a run-time error and *Stop*. Traditionally, the outputting process would be rescheduled and continue executing, assuming the inputter had successfully participated in the communication – which is untrue if the tag was unhandled. Normally, this is not a problem since the run-time error generated by the inputting process causes the program to abort. However, the OS-grade run-time loaded processes [29] do allow an occam process to generate a run-time error and *Stop* without shutting down the whole system, in which case this problem becomes much more relevant.

An experimental compiler option is available which causes tagged-protocol inputs to always perform an extended input on the tag value, if that protocol contains *data-less* tags which the inputting process does not handle. In *stop error-mode* (not handled properly yet), where processes just *Stop* rather than abort with run-time errors, this corrects a long-standing (although minor) delinquency of the occam language.

6 Dynamic Process Creation (the FORK)

The FORK is a way of launching a dynamic process which runs parallel to the dispatching process. The early ideas about FORK were to allow an arbitrary process to be spawned (that process being indented under the FORK). This was causing too many headaches in the implementation however, so a more restricted approach has been taken for now: the parallel creation of a PROC instance. This provides a nice way of giving the launched process an initial state – its parameters. The more general FORK would need to provide a way of handling scoping and parallel usage for free variables in the FORKed process. Controlling this through the parameters of a PROC is much simpler.

The lifetime of a FORKed PROC and its dispatching process are controlled through a special FORKING process constructor. This acts as a barrier which ensures any FORKed processes are complete before leaving the FORKING block. For example, here is part of much simplified code from a *dynamic* version of the occam web-server [15]:

```
PROC fe.proc (VAL INT n, D.CONN conn, SHARED C.CONN! to.sw)
  ...
:

PROC fe.farm (CHAN D.CONN in?, SHARED C.CONN! to.sw)
  D.CONN local:
  FORKING
    INITIAL INT c IS 0:
    WHILE TRUE
      SEQ
        in ? local
        FORK fe.proc (c, local, CLONE to.sw)
        c := c + 1
:
```

The 'fe.farm' process sits in a loop accepting D.CONN's (connection in the web-server) from its 'in' channel. For each D.CONN received, an instance of 'fe.process' is created. These processes are actually pooled for recycling – see [1] for details on this.

The need for the 'FORKING' block may not seem immediately obvious, here less so than in other cases, but the implementation requires it (section 6.2). A more obvious case is where we share data with FORKed processes using the "#PRAGMA SHARED name" compiler directive (to turn off usage and alias checking). In these cases, we must guarentee (completely) that shared variables remain in scope for the whole lifetime of any FORKed process.

6.1 Semantics of FORK Parameters

Unlike non-FORKed PROCs, whose parameters follow a *renaming* semantics, the parameters in a FORKed PROC have to follow a *channel communication* semantics. This has different semantic effects to an ordinary PROC call. We allow only the following types of parameters:

- VAL data-types: these are *copied* into the FORKed process, regardless of size. This differs from traditional VAL parameter passing, which will abbreviate (rename) items larger than 4 bytes (1 word).
- MOBILE data-types and MOBILE channel-type-ends: these are *moved* into the FORKed process – i.e. the FORKING process loses them. If the FORKING process does not want to lose them, it must pass a CLONEd argument[4].
- Reference parameters: which have been explicitly *shared* (with a compiler #PRAGMA) and which are declared *outside* the FORKING block. This ensures that they remain in scope for the lifetime of a FORKed PROC. Variables declarared *within* the FORKING block may not be passed by reference.

The copying of VAL data-types is *required*, as we wish to allow the code on the left (below):

```
FORKING                              INT x:
  INT x:                             SEQ
  SEQ                                  x := 42
    x := 42                            PAR
    FORK P (x)   -- VAL param            P (x)        -- VAL param
    x := x + 1                           SEQ
    Q (x)                                  x := x + 1
                                           Q (x)
```

The left code (above) is not equivalent to the right code (which is, of course, illegal since 'x' is both read and assigned in parallel). The FORKING block given above (left) is in fact equivalent to the following processes:

```
CHAN INT c:
PAR

  INT x:
  SEQ
    x := 42
    c ! x
    x := x + 1
    Q (x)

  INT x:
  SEQ
    c ? x
    P (x)
```

Semantically, there is no difference between parameter passing VALs by communication (as we have done here), and parameter passing using INITIAL formals. INITIAL formal parameters [10, 30] should be used, but these are not currently supported by the compiler.

[4]For channel-types, only explicitly SHARED ends may be CLONEd

Some more interesting FORKs (i.e. with a loop), can also be expressed in our extended occam. For example, the following two (columns of) processes are equivalent:

```
FORKING
  WHILE TRUE
    SEQ
      P ()
      FORK foo (42, 99)
      Q ()
```

```
RECURSIVE PROC dispatch (CHAN BOOL c?)
  SEQ
    BOOL any:
    c ? any
    PAR
      foo (42, 99)
      dispatch (c?)
:

CHAN BOOL c:
PAR
  dispatch (c?)

  WHILE TRUE
    SEQ
      P ()
      c ! TRUE
      Q ()
```

6.2 Implementation

The implementation of the FORK is reasonably simple and is done in such a way that all the memory handling is done in unit-cost time. In practice, it is fairly similar to the method used to implement self-recursion in KRoC/Linux (a description of which can be found in [1]). The FORKING construct introduces a special declaration, '$fork.barrier', of the wholly internal type 'BARRIER' [7]. Since only one process will ever actually synchronise on this barrier, handling for multiple synchronisers can be skipped, simplifying the implementation somewhat. The BARRIER gets placed in the process's workspace, along with other local variables. Figure 8 shows the layout of the barrier in memory. The '*Fptr*' field is used to hold the FORKING process when it finished (assuming there are still FORKed processes active). The '*count*' field indicates how many processes are enrolled on the barrier.

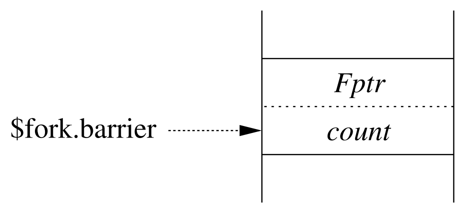

Figure 8: Layout of the FORKING barrier in workspace

When the FORKING block starts, the barrier is initialised such that *Fptr* is *null* and *count* is 0 (no enrolled processes). In the ETC [31] output of the compiler, this is just:

```
NULL
ST    $fork.barrier + 1
LDC   0
ST    $fork.barrier + 0
```

Each time a FORK happens, the memory required for the new process is allocated from the free-lists. This is for both *workspace* and *vectorspace*. The space allocated for the new workspace is the process's workspace requirements, plus any space required for copies of `VAL` reference parameters, plus a small bit for housekeeping. If the process requires *mobile-space* [8, 9] then, although we can allocate it from the free-lists, freeing it is not allowed. This is because pointers into its mobilespace may have migrated into the mobilespace of other processes, and in turn we will aquire pointers into other mobilespaces. An implementation of "*mobilespace fixup*" is possible, such that it copies data around to regain all its references, but this would involve run-time overheads – we would need to track mobilespace creation, have a mechanism for locating the *shadow-slot* of a given mobile and have a mechanism for waiting for processes to free any in-use bits of our mobilespace.

Instead of just not freeing mobilespace, we free it to a private free-list which is allocated in the mobilespace of the `FORKING` process. Figure 9 shows this arrangement, after the '`data`' variable has been brought into scope, but before '`thing`' has been forked off. The '`data`' pointer may not necessarily be pointing at the mobilespace indicated, since it will move each time '`thing`' is forked (it is passed as a parameter); it may make its way back at a later stage however.

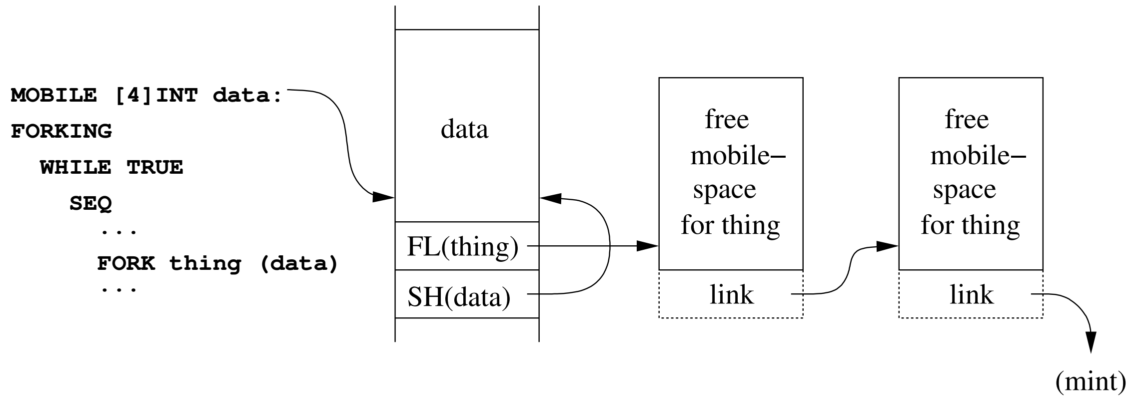

Figure 9: Layout of free mobilespace for `FORK`ed '`thing`'

At the point where the `FORK` occurs in figure 9, there are two free blocks of mobilespace available. If the "FL(thing)" pointer is *mint* (most negative integer) then a new block is allocated from the system. The word below the allocated block is usually used by the system in order to free the block to the right free-list. Since this memory is not going back to the regular free-lists, it is recycled to contain a 'link' field which links the free blocks together. Additionally, when the (dynamic) mobilespace is in use, this link field points at the 'FL(thing)' slot, so that we can free the memory when done. This is not entirely dissimilar to the method used to implement mobilespace for recursive processes.

Once the new workspace has been allocated, a link to the `BARRIER` is stored and the parameters are loaded in as appropiate (which will include vectorspace and mobilespace if needed). The new process is enrolled on the barrier, which is a simple increment of *count*, and then added to the run-queue for scheduling. As noted in the previous section, some parameters require special treatment, such as copying `VAL` parameters or incrementing the reference-count for a `SHARED` channel-type variable. All this is done when the parameters are loaded into the target workspace.

Once the FORK finishes, the first job is to *resign* from the barrier, which might need to reschedule the blocked FORKING process if it finished first. The code generated for doing this is:

```
    -- decrement count field
    LD    $fork.barrier + 0     -- load count
    ADC   -1                    -- minus 1
    ST    $fork.barrier + 0     -- store count

    -- any waiting process ?
    LD    $fork.barrier + 1     -- load pointer
    NULL                        -- load null
    DIFF                        -- subtract
    CJ    :L1                   -- jump if 0

    -- yes, count 0 ?
    LD    $fork.barrier + 0     -- load count
    CJ    :L0                   -- jump if 0
    J     :L1                   -- jump

:L0
    -- yes, reschedule it
    LD    $fork.barrier + 1     -- load pointer
    NULL                        --   (load null)
    ST    $fork.barrier + 1     --   (clear pointer)
    RUNP                        -- run process

:L1
```

After this, any memory used by the finishing FORKed process is returned to the appropiate free-list. Although the '`$fork.barrier`' really lives in the workspace of the FORKING process, we saved a link to it while setting up for the FORK. Usually, a process terminates itself by calling 'STOPP' (simple) or 'ENDP' (used with PARs). As it turns out, these are not much use if we wish the process to free its own workspace before terminating. Trying to 'MRELEASE' after the 'STOPP' is a no-op and 'MRELEASE' before 'STOPP' would result in a block of free memory being used (albeit briefly). Neither of these is really an option so another instruction has been added – 'MRELEASEP'. This takes an 'adjustment' argument which is added to 'Wptr' (the workspace pointer) before freeing. Unlike 'STOPP', this instruction does not need to save the return-address since it will never be run again.

When the FORKING block finishes, it attempts to synchronise on the barrier. If it sees *count* as 0, it simple does nothing and continues with whatever follows the FORKING. Otherwise it suspends itself in the Fptr space:

```
    LD    $fork.barrier + 0     -- load count
    CJ    :L2                   -- jump if 0
    LDLP  0                     -- load Wptr
    ST    $fork.barrier + 1     -- store pointer
    STOPP                       -- deschedule
:L2
```

7 Process Priority

A major requirement of real-time control applications is a set of cyclic processes – one for each *control-law* – managed so that each process completes each cycle within a fixed time. The rate at which each process cycles will be constant, but will generally be different for different processes.

Transputer hardware [32, 33] supported two levels of priority, low and high, with fast pre-emptive scheduling. The original KRoC [6] only supported a single level of priority, quietly implementing 'PRI PAR' as just 'PAR'. This is not sufficient to manage securely more than one such 'control-law' per occam program – even at very low processor loadings. Efficient classical solutions (e.g. *rate-monotonic* or *deadline* scheduling [34]) require multiple and time-varying priorities. KRoC now provides 32 levels of priority.

There are ways of providing priority scheduling without having it as part of the run-time system however, as has been done in [35] and [36]. For KRoC/Linux however, we are at the mercy of the underlying operating-system and the way in which it performs priority scheduling between OS processes. There are ways of forcing *run-to-completion* behaviour for OS processes (*fifo-scheduling*), but at the expense of other OS processes (including interrupt handlers) and the requirement for *superuser* privileges. We are investigating *Raw Metal occam* operating environments (RMOX) in which occam systems run without any OS support and overheads, and for which all scheduling is under the total control of the KRoC kernel (with minor modifications for this environemnt).

Standard CSP [3] does not include a treatment of process priority, and omits PRI ALT, which has existed in occam for years. Even the denotational semantics for occam [37, 38] expressly omit any treatment of priority. $\mathcal{CSPP}$ [39, 40, 41, 42] addresses this deficiency by providing a well-defined semantics for PRI PAR and occam priority issues in general.

7.1 The Language Binding

Rather than implement priority in terms of PRI PAR, we have gone for a more general – but lower level – approach. The current implementation supports 32 distinct levels of priority, 0 being the highest and 31 being the lowest. The number of priority levels is limited in order to be efficient in the implementation, but is theoretically extensible. To inspect or change its own priority, a process may use the following compiler built-ins:

```
INT FUNCTION GETPRI ()
PROC SETPRI (VAL INT p)
PROC INCPRI ()
PROC DECPRI ()
```

The use of 'SETPRI'/'INCPRI'/'DECPRI' within a FUNCTION body is not allowed – to prevent non-determinism in the resulting priority when evaluating expressions. 'GETPRI' is wholly non-side-effecting, so can be used safely in expressions. The run-time implementation quietly ignores out-of-range values, mapping them to the lowest and highest priorities as appropiate. 'INCPRI' is really a shorthand way of writing:

```
SETPRI (GETPRI() - 1)
```

and similarly for 'DECPRI', which is:

```
SETPRI (GETPRI() + 1)
```

A process may change its priority arbitrarily – it cannot change the priority of any other process. Changing from a low to a high priority (decreasing priority level 'p') will generally always succeed immediately, although there is a possibility of descheduling (discussed in section 7.2). A change from a high to a low priority (increasing priority level 'p') will result in a reschedule if another process is waiting at the target priority level or higher.

7.2 The Implementation

The implementation of process priority has attemped to be as non-intrusive as possible, in order to minimize any loss of program performance. The basic ideas follow those proposed by Ploeg et. al. [16].

A significant modification is the introduction of a *process priority slot* in the process workspace. This has been inserted between the 'Link' and 'Pointer' fields, as shown in figure 10. The "below workspace" slots are only used when a process is blocked – i.e. not on any run-queue.

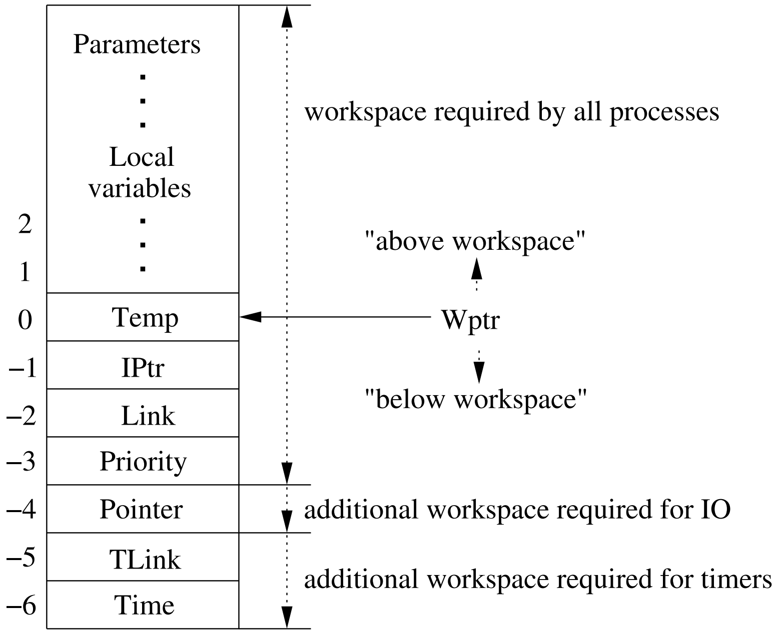

Figure 10: New process workspace layout

The run-time kernel (a heavily modified CCSP [43]) maintains 32 seperate run-queues, one for each priority level. Priority is managed through the use of two kernel variables, *PPriority* and *PState*. *PPriority* holds the priority level for any running process and is what occam processes retrieve when they call '`GETPRI()`'. *PState* is a bit-field indicating at which priority levels processes exist. This can be scanned in a single-cycle instruction (on i386 architectures at least) to determine the highest priority of any runnable process. A similar approach is used to handle priority in MESH [44, 45].

At run-time, an additional synchronisation flag [6] is used to indicate the presence of a higher-priority process. Once this flag is set, the scheduler will deschedule the current process and reschedule at the first opportinity. This flag is needed because there are certain cases where a process becomes runnable at a higher priority, but where the current process cannot be immediately descheduled.

Generally speaking, a process can only be added to the run-queue through the use of the '`STARTP`' and '`RUNP`' instructions. '`STARTP`' is used to create a new process, which is added to the run-queue for scheduling. The newly created process inherits the current priority, so goes on the current run-queue. As such, the '`STARTP`' instruction can still be in-lined [46] with very little extra cost for priority. '`RUNP`' is slightly trickier, since the process being resumed has its own priority stored in its 'Priority' workspace slot. This is examined and either placed on the appropiate run-queue, or scheduled immediately, storing the current process on its run-queue. The rescheding of processes following communication is also done this way (it is effectivly a `RUNP` of the process which arrived at the channel first).

A process of a higher priority may also become runnable due to an external signal or the completion of a blocking system-call [47]. For blocking system-calls, this means reducing the $O(1)$ collection operation into an order $O(n)$ operation, where n is the number of completed system-calls. This is because we must examine each completed system-call in order to place it on the right run-queue, or to schedule it immediately (because it was of a higher priority than any other runnable processes). Typically, n is 1 or 2, so the loss associated with handling priority is pretty small.

Once an event occurs such that a higher priority process becomes runnable, we need to ensure that it will be rescheduled. Many times, this is not a problem since the scheduler will be entered immediately (by the process causing that event) and the appropiate priority-related actions taken. However, when the event is caused by some external action (e.g. a hardware interrupt or timeout), there can be a problem. For example:

```
PAR
  --{{{  high priority process
  SEQ
    SETPRI (0)
    ...  block waiting for an external event
  --}}}
  --{{{  low priority process
  SEQ
    SETPRI (1)
    WHILE TRUE
      ...  pure background computation
  --}}}
```

Here the high-priority process will be scheduled first and immediately blocks waiting for some external event. The scheduler then schedules the low-priority process, based on the absence of anything else runnable. This goes into an infinite background computation loop. When the event occurs, the relevant scheduler synchronisation flag will be set, however the scheduler is never entered again to be able to notice this and resume the blocked process. Not very desirable.

The solution to this was constructed by adding a compile-time option to make any generated *backward jumps* rescheduling points. This conforms to the transputer implementation [33], which specifies the jump (‘`J`’) instruction to be a potential descheduling point. The KRoC option (‘`-P`’) is passed to the translator (`tranx86` [46]), which inserts the relevant code to reschedule if needed. A series of inline checks is generated at such points, which check the scheduler sync flags and timer-queue for activity.

Translator	Channel cost (`INT` communication)	Process startup/ shutdown cost
‘`tranpc`’ (old KRoC)	233	196
‘`tranx86`’	112	52
‘`tranx86`’ (inlining)	52	28
‘`tranx86`’ (priority)	120	108
‘`tranx86`’ (pri + inline)	77	79
‘`tranx86`’ (pri + ‘`-P`’)	119	116
‘`tranx86`’ (pri + inline + ‘`-P`’)	75	67

Table 2: Results for the ‘`commstime`’ benchmark on an 800 MHz Pentium-3. *Values are in nano-seconds.*

Table 2 shows results for the '`commstime`' benchmark program [6], which measures the cost of communication and process creation/shutdown. The process network for `commstime` is shown in figure 11. These times were measured on a moderately idle 800 MHz P3. The difference in loop times for `CHAN INT` communication, with and without priority, is not significant (8 ns), but the process startup/shutdown time doubles (to 108 ns). Enabling *inlining* reduces the process startup/shutdown time to 79 ns – an increase of 51 ns. Turning on the '`-P`' (reschedule on jump) option with inlining actually has a slight positive impact, attributed to cache noise on the benchmark.

Overall though, these overheads remain at the order of 100 nano-seconds (or 100 cycles), even though the kernel is now supporting 32 levels of priority.

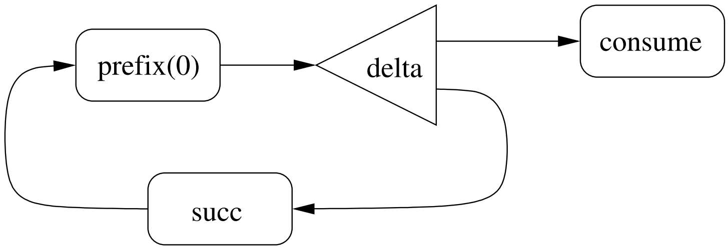

Figure 11: Process network for the '`commstime`' benchmark program.

7.3 Benchmarking Priority Handling

Figure 12 shows the process network and code for a simple priority benchmark program. The benchmark is comprised of three processes, two interleaving producers and a consumer. The consumer is run at the lowest priority level (31) while the consumers alternate between priority levels 1 through 4. The arrangement of priority and communication in this program forces the scheduling to happen in a deterministic way, although *in general* one cannot use priority to guarantee determinism.

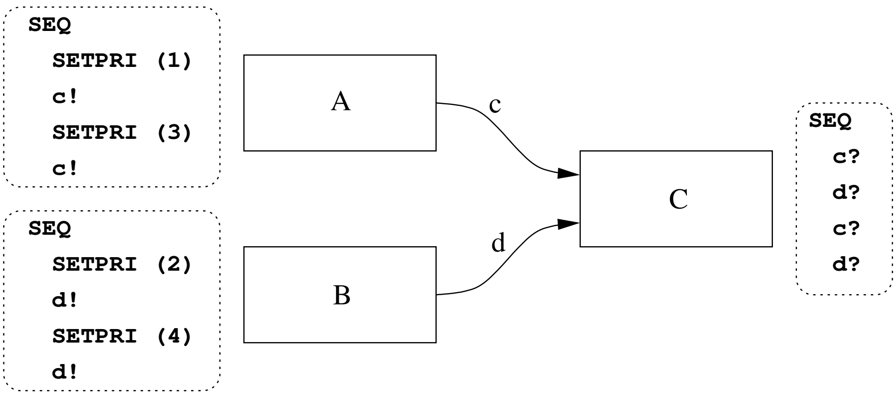

Figure 12: Priority benchmark process network and loop body code.

Both the `A` and `B` processes sit in loops changing priority and communicating with the `C` process. At the point where the loops start, `A` and `B` are blocked in channels '`c`' and '`d`' at priorities 1 and 2 respectively. Process `C` at priority level 31 is the only runnable process, which starts by commnicating on '`c`', thereby waking up `A`, which gets rescheduled immediately because it is of a higher priority.

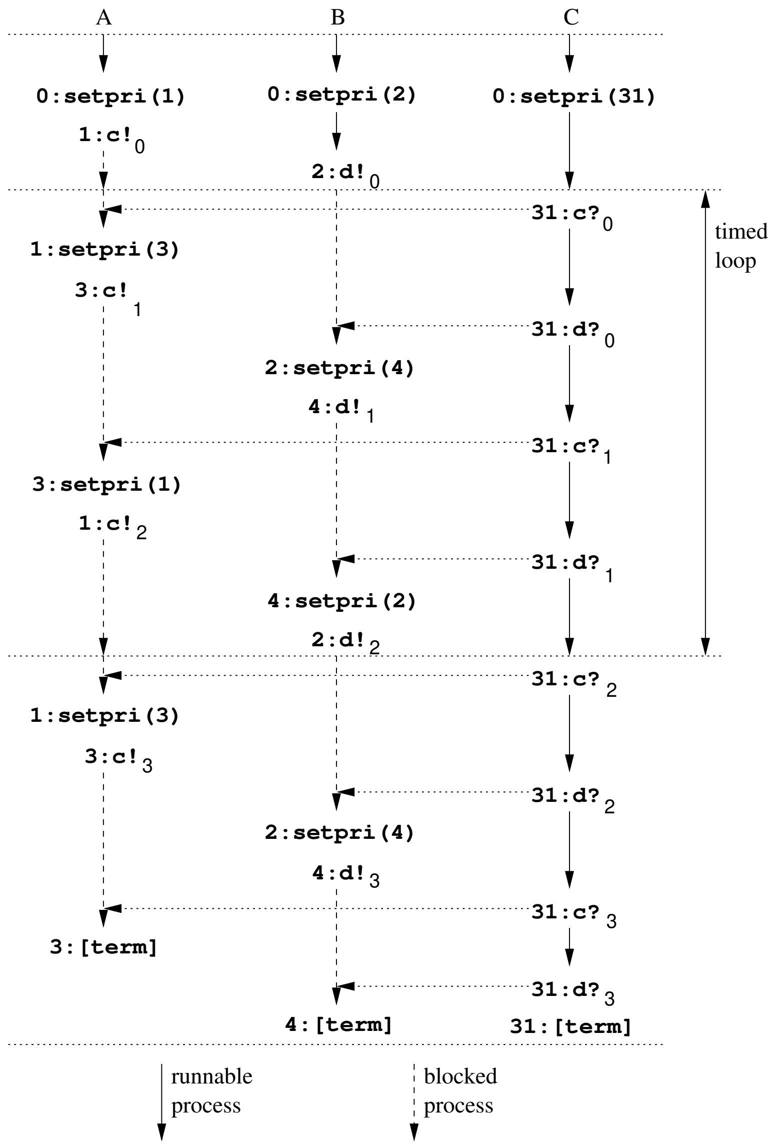

Figure 13: Execution trace for priority benchmark program.

Figure 13 shows the execution trace of the benchmark, indicating the timed (and looping) section. The priority overheads are calculated by subtracting the time required for a priority-free version of the loops from the prioritised version. There are a total of 8 context switches here, four for rescheduling when a process blocks in a channel (A and B processes), and four for rescheduling a higher-priority process with which process C communicates. There are also other overheads associated with changing priority, since the current run-queue must be saved and a new one loaded. The non-prioritised version of the loops use an average of 8 context switches, the exact number depends on the scheduling order of processes. It is also sensitive to the policy of rescheduling blocked processes – i.e. whether we continue running, it continues running, or neither continue running.

The priority overhead for this benchmark loop is 752ns, measured on an 800 MHz Pentium 3. Interestingly, with the '-p' flag (check sync flags and timer on backward jumps), the overhead is reduced to 728ns, again attributed to cache effects. Overall, the run-queue is changed 12 times in the prioritised version, giving an average overhead of 63ns for a priority-level change (around 50 machine cycles).

8 Conclusions and Future Work

At the time of writing, the extensions presented here are almost ready for release. There are one or two things which are currently unhandled by the compiler, but these will be finished off in version 1.3.3 of KRoC/Linux. A new experimental occam web-server (which uses FORKs and channel-types) is currently running live at [15] and successfully serving pages.

It is also hoped that improvement in the translator's handling of priority will yield better benchmark times for '`commstime`' (Table 2). This is especially true for inlining, where the presence of priority often causes non-inline translations for some instructions.

It has been pointed out that allowing run-time failures (with OS-grade loadable occam processes [29]) presents the possibility of dynamic memory being "lost" (inside the terminated process or processes). Although it is not currently implemented, it is possible to recover the memory. The solution is also required for correct loading/saving of these loadable occam processes, although the current (somewhat naïve) implementation of this works. The required addition is in theory quite simple – generate in-line tables of workspace offsets for channels and dynamic-pointers. This will allow the run-time system to free dynamic memory used by an *evicted* process. It is hoped to implement this in the near future, since without it saving and restoring loadable processes has the potential to go wrong.

Another aspect of dynamic occam not covered here is the issue of *mobile processes* (or agents). These are processes which contain a seperate state in addition to their local workspace and vectorspace, and which can be communicated around a process network carrying that state with them. Such capabilities already exist for JCSP [48, 49]. For occam, there is still a slight issue with appropiate syntax for this, since the persistent-state and initialisation code need to be seperated from the `PROC` contents slightly. However, the initialisation code should *not* be in the scope of the `PROC` parameters, since they are supplied each time the mobile process is run.

Finally, as mentioned at the start of section 7, we are investigating *Raw Metal occam* operating environments (RMOX) in which occam systems run without any OS support and overheads, and for which all scheduling is under the total control of the KRoC kernel. We are using the Flux OSKit [50] to provide the boot mechanism and access to a flat (physical) memory space, beyond that very little else of the OSKit is used. This work is being done in collaboration with Brian Vinter from the University of Southern Denmark.

9 Acknowledgements

The authors would like to thank EPSRC for funding this work (in the form of a research studentship), and the anonymous reviewers who provided useful and detailed feedback on an earlier revision of this work. Also many thanks to the people who have been patient with the various new features of KRoC/Linux, submitting valuable bug reports and providing useful thoughts, in particular: David Wood, Adrian Lawrence, Hiroshi Nakahara and Mario Schweigler.

References

[1] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part II. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 363–380, IOS Press, Amsterdam, The Netherlands, September 2002.

[2] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

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

[4] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: `http://wotug.ukc.ac.uk/parallel/occam/documentation/`.

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

[6] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166. World occam and Transputer User Group, IOS Press, Netherlands, March 1996. ISBN: 90-5199-261-0.

[7] Peter H. Welch and David C. Wood. Higher Levels of Process Synchronisation. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 104–129, Amsterdam, The Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press. ISBN: 90-5199-336-6.

[8] F.R.M. Barnes and P.H. Welch. Mobile Data Types for Communicating Processes. In *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, volume 1, pages 20–26. CSREA press, June 2001. ISBN: 1-892512-66-1.

[9] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Majid Mirmehdi Alan Chalmers and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 243–264, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[10] Geoff Barrett. occam 3 Reference Manual. Technical report, Inmos Limited, March 1992. Available at: `http://wotug.ukc.ac.uk/parallel/occam/documentation/`.

[11] Henk L. Muller and David May. A simple protocol to communicate channels over channels. In *EURO-PAR '98 Parallel Processing, LNCS 1470*, pages 591–600, Southampton, UK, September 1998. Springer Verlag.

[12] David May and Henk Muller. Copying, Moving and Borrowing semantics. In Majid Mirmehdi Alan Chalmers and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 15–26, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[13] F.R.M.Barnes P.H.Welch, J.Moores and D.C.Wood. The KRoC Home Page, 2000. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`.

[14] F.R.M. Barnes. Various extensions to the occam compiler, 2001. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/occ21-extensions.html`.

[15] F.R.M. Barnes. The occam Web-Server Home Page, 2000. Available at: `http://wotug.ukc.ac.uk/ocweb/`.

[16] E. Ploeg, J. P. E. Sunter, A. W. P. Bakkers, and H. W. Roebbers. Dedicated multi-priority scheduling. In Roger Miles and Alan Chalmers, editors, *Proceedings of WoTUG-17: Progress in Transputer and Occam Research*, volume 38 of *Transputer and Occam Engineering*, pages 18–31. IOS Press, The Netherlands, April 1994. ISBN: 90-5199-163-0.

[17] P.H. Welch. Five Essays on Occam. *Occam User Group Newsletter*, 2, January 1985. Also Internal Report, Training Department, GEC Avionics Ltd., Airport Works, Rochester, KENT ME2 1XX.

[18] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent at Canterbury, August 1998.

[19] P.H. Welch, G.R.R. Justo, and C.J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press, Netherlands. ISBN 90-5199-140-1.

[20] David May and Henk L. Muller. Using Channels for Multimedia Communication. Technical report, University of Bristol, Department of Computer Science, February 1998.

[21] T.S. Locke. Towards a Viable Alternative to OO – extending the occam/CSP programming model. In Majid Mirmehdi Alan Chalmers and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 329–349, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[22] D.C.Wood and J.Moores. User-Defined Data Types and Operators in occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 121–146. WoTUG, IOS Press, the Netherlands, April 1999. ISBN: 90-5199-480-X.

[23] P.H.Welch. Process Oriented Design for Java – Concurrency for All. In *PDPTA 2000*, volume 1, pages 51–57. CSREA Press, June 2000. ISBN: 1-892512-52-1.

[24] G.E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[25] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York, 1996, reprint 1997.

[26] Per Brinch Hansen. Efficient Parallel Recursion. *ACM SIGPLAN Notices*, 30(12):9–16, December 1995. Reprinted in: *The Origin of Concurrent Programming*, edited by Per Brinch Hansen, pp. 525-534, Springer, ISBN 0-387-95401-5. 2002.

[27] Per Brinch Hansen. Efficient Parallel Recursion. In Per Brinch Hansen, editor, *The Search for Simplicity: Essays in Parallel Programming*, pages 509–518. IEEE Computer Society, Los Alamitos, California, 1996. chapter 25.

[28] P.H. Welch. An occam Approach to Transputer Engineering. In *Proceedings of the 3rd. Conference on Hypercube Concurrent Computers and Applications*, Pasadena, California, USA, January 1988. ACM, ACM Conference Proceedings.

[29] F.R.M. Barnes. Dynamic occam Processes, 2000. Fringe-session presentation at CPA-2000, available at: `http://frmb.home.cern.ch/frmb/pubs/dynoccam-slides.ps`.

[30] James Moores. *The Design and Implementation of occam/CSP Support for a Range of Languages and Platforms*. PhD thesis, The University of Kent at Canterbury, Canterbury, Kent. CT2 7NF, December 2000.

[31] M.D.Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 187–198, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[32] P.W. Thompson M.D. May and P.H. Welch. *Networks, Routers and Transputers*, volume 32 of *Transputer and occam Engineering Series*. IOS Press, 1993.

[33] Inmos Limited. *The T9000 Transputer Instruction Set Manual*. SGS-Thompson Microelectronics, 1993. Document number: 72 TRN 240 01.

[34] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[35] J.P.E. Sunter, K.C.J. Wijbrans, and A.W.P. Bakkers. Cooperative Priority Scheduling in Occam. In H.S.M. Zedan, editor, *Proceedings of the 13th occam User Group Technical Meeting: Real-Time Systems with Transputers*, Transputer and Occam Engineering, pages 175–185. IOS Press, The Netherlands, September 1990. ISBN: 90-5199-041-3.

[36] P.H. Welch. Multi-Priority Scheduling for Transputer-Based Real-Time Control. In H.S.M. Zedan, editor, *Proceedings of the 13th occam User Group Technical Meeting: Real-Time Systems with Transputers*, Transputer and Occam Engineering, pages 198–214. IOS Press, The Netherlands, September 1990. ISBN: 90-5199-041-3.

[37] A.W. Roscoe M.H. Goldsmith and B.G.O. Scott. Denotational Semantics for occam2, Part 1. In *Transputer Communications*, volume 1 (2), pages 65–91. Wiley and Sons Ltd., UK, November 1993.

[38] A.W. Roscoe M.H. Goldsmith and B.G.O. Scott. Denotational Semantics for occam2, Part 2. In *Transputer Communications*, volume 2 (1), pages 25–67. Wiley and Sons Ltd., UK, March 1994.

[39] A.E.Lawrence. Extending CSP. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 111–131, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[40] A.E. Lawrence. Successes and Failures: Extending CSP. In Majid Mirmehdi Alan Chalmers and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 49–66, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[41] A.E. Lawrence. CSPP and Event Priority. In Majid Mirmehdi Alan Chalmers and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 67–92, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[42] Adrian Lawrence. Acceptances, Behaviours and Infinite Activity in CSPP. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 17–38, IOS Press, Amsterdam, The Netherlands, September 2002.

[43] J.Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[44] R.W. Dobinson M. Boosten and P.D.V. van der Stok. Fine-Grain Parallel Processing on Commodity Platforms. In *WoTUG 22*, volume 57 of *Concurrent Systems Engineering*, pages 263–276. IOS Press, the Netherlands, April 1999. ISBN: 90-5199-480-X.

[45] R.W. Dobinson M. Boosten and P.D.V. van der Stok. MESH: MEssaging and ScHeduling for Fine-Grain Parallel Processing on Commodity Platforms. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'1999)*. CSREA press, June 1999. ISBN: 1-892512-15-7.

[46] F.R.M. Barnes. tranx86 – an Optimising ETC to IA32 Translator. In Majid Mirmehdi Alan Chalmers and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 265–282, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[47] F.R.M. Barnes. Blocking System Calls in KRoC/Linux. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures*, volume 58 of *Concurrent Systems Engineering*, pages 155–178, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[48] P.H.Welch, J.R.Aldous, and J.Foster. CSP networking for java (JCSP.net). In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002. ISBN: 3-540-43593-X.

[49] P.H. Welch and B. Vinter. Cluster Computing and JCSP Networking. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 213–232, IOS Press, Amsterdam, The Netherlands, September 2002.

[50] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997. Software available from: `http://www.cs.utah.edu/flux/oskit/`.