Communicating Process Architectures – 2000
P.H. Welch and A.W.P. Bakkers (Eds.)
IOS Press, 2000

Blocking System Calls in KRoC/Linux

Frederick R.M. Barnes
Computing Laboratory, University of Kent, Canterbury, KENT. CT2 7NF
(frmb2@ukc.ac.uk)

Abstract. This paper describes an extension to Kent Retargetable occam Compiler [1] (KRoC), which enables the execution of a blocking call, without blocking the occam-kernel. This allows a process to make a blocking system call (eg, `read`, `write`), without blocking processes running in parallel with it. Blocking calls are implemented using Linux clones which communicate using shared memory, and synchronise using kernel level semaphores. The usefulness of this is apparent in server applications with a need to handle multiple clients simultaneously. An implementation of an occam web-server is described in section 5, which uses standard TCP sockets via an occam socket library. The web-server comes with the ability to execute CGI scripts as well as dispensing static pages, which demonstrates some level of OS process management from within occam.

However, this mechanism is not limited to blocking in the Linux kernel. On multiprocessor machines, the clones are quite free to be scheduled on different processors, allowing computationally heavy processing to be performed aside the occam world, but still with a reasonable level of interaction with it. Using them in this way provides a coarse-grained level of parallelism from within the fine-grained occam world.

1 Introduction and Motivation

The standard way of communicating with a KRoC occam program is via the use of the three byte channels passed to the top-level process. These are tied to three simple processes which provide access to the UNIX `stdin`, `stdout` and `stderr` streams. As these are `BYTE` channels, each communication causes a context switch (light-weight), as well as associated system call to read or write the byte (heavy-weight).

There are two possible alternatives where more elaborate interaction with the outside world is required. The first is to use the `hostio` and `hostsp` libraries [2]. These provide enough IO facilities to get jobs done. For example, `tranpc`[1] uses these libraries to read and write files. The second alternative is to write your own in the C world and provide occam with an interface to call it.

Writing functions in C has advantages, such as access to all the facilities of the operating-system, in this case Linux. Its usefulness is limited though, as any system call which blocks in the kernel will suspend the Linux process who made it, in this case, the entire occam program. A naive solution would be to let each occam process run as a separate Linux process, communicating via standard kernel mechanisms. Of course, this is unfeasible since the overheads are colossal when compared to the sub microsecond performance of KRoC. A

[1]Extended transputer code[3] to Intel 386 ELF object code converter

multiprocessor KRoC would relax the problem slightly, as one instance of the occam-kernel could block, while the others (on different CPUs) carried on scheduling occam processes. This is not feasible either, since there is a limit on the number of simultaneous blocking calls possible; one less than the number of running kernels.

The solution presented here uses Linux `clone()`s to hand off the blocking call to another Linux process. If this call does block, then only the clone in which it is running will block; occam processes running in parallel with the blocking call will continue to be scheduled. Using this method, the number of simultaneous blocking calls possible is limited only by the user's process limits, currently 8192. In actual fact we set the limit to a lower value of 256. This is mainly because we do not want an awry occam program thrashing Linux. The value can be increased however, and the run-time system re-compiled if there is a specific need for more.

Since the occam compiler performs parallel usage checking, any unsafe parallel usage of a blocking call will be rejected. It does not, however, guarentee deadlock, livelock or starvation freedom; that is up to the programmer.

2 The occam/C Interface

External functions are introduced to occam by the use of the `EXTERNAL` compiler directive. The standard way of calling an external C function is to declare it as a `PROC` to occam, then provide the necessary C function by linking it in. Details on how to do this (including parameter passing mechanisms) are given in [4]. A few example `EXTERNAL` declarations might be:

```
#PRAGMA EXTERNAL "PROC C.foo (VAL INT x, VAL INT y, INT z) = 0"
#PRAGMA EXTERNAL "PROC C.bar ([]BYTE arry, INT len) = 0"
```

When `tranpc` sees a reference to an external function starting with '`C.`', it generates a special calling sequence, using the external function '`_call_C_interf`'. This function is provided as part of the standard run-time occam system, in this case CCSP [5].

Blocking calls are declared in a similar way, but instead of starting with '`C.`', they start with '`B.`' or '`BX.`'. The calls starting with '`B.`' identify a normal blocking call; those starting with '`BX.`' indicate a blocking call on whose termination occam may `ALT`. Terminatable blocking calls are discussed in section 3.5. `tranpc` has been modified to spot these '`B.`' and '`BX.`' calls, generating similar code to external C calls, but calling the interface functions '`_call_B_interf`' and '`_call_BX_interf`' respectively. The "`= 0`" on the end of an external C declaration indicates the number of workspace words required for the external procedure. As C functions do not use any occam workspace, (except for the parameters they were passed which has already been accounted for), this is set to zero. External blocking calls on the other hand need two words of workspace. These are the two words used by the scheduler when the process is waiting on the run queue (instruction pointer and queue link).

For the above example external declarations '`C.foo`' and '`C.bar`', the C functions '`_foo`' and '`_bar`' would need to implemented. Blocking versions follow a similar naming scheme in such a way that one C function can implement all three occam versions. It is quite legitimate, and often useful, to be able to write:

```
#PRAGMA EXTERNAL "PROC C.foo (VAL INT x, VAL INT y, INT z) = 0"
#PRAGMA EXTERNAL "PROC B.foo (VAL INT x, VAL INT y, INT z) = 2"
#PRAGMA EXTERNAL "PROC BX.foo (CHAN OF INT c, VAL INT x,
                      VAL INT y, INT z) = 2"
```

Only one C function is required for all of these, '`_foo`'. The extra parameter in the '`BX.foo`' call is used for termination (section 3.5).

3 The Clones

A *clone* process on Linux is effectively another OS-level process, but can share virtual memory, file descriptors and/or file-system information with its parent. Clones maintain their own stack and process context, as it does not make sense to share these. Clones are created through the use of the `clone()` system call, which takes arguments describing what wants to be shared, the clone's stack, the function where the clone starts, and an arbitrary parameter for that function.

3.1 Starting it all

To communicate between the clones and the occam-kernel process, a number of shared variables are used. (In actual fact, everything in the heap is shared, but the clones only read/write certain variables, and carefully at that):

- '`bsc_thread dispatching`' is used to pass information about the new blocking call between the occam-kernel and a clone (section 3.3).
- '`spl_t dispatch_lock`' is the spin-lock used to protect the above structure. It is locked by the occam-kernel, and released by the clone.
- '`pid_t clone_pids[]`' holds the process IDs (PIDs) of the clones. The first clone places it's PID at index 0, the second clone at index 1, and so on. This is used by the clones during termination (section 3.5).
- '`bsc_thread *clone_arry[]`' holds a pointer to each clone's `bsc_thread` structure, as described in the next section. The clones use this array, in conjunction with the 'tt clone_pid[]' array, to find themselves during termination.
- '`word *q_fptr, *q_bptr`' are the queue pointers on which finished clones place their occam processes. It is described in section 3.4.
- '`int num_queued`' is a counter indicating how many occam processes are on this queue.
- '`spl_t queue_lock`' is the spin-lock used to protect the three queue and counter variables above.

3.2 Creating new clones

Each clone gets created with a stack size of just under 128 kilo-bytes. At the bottom of this stack is a structure which holds the state of the clone:

```
struct _bsc_thread {
    int pid;                       // clone's process ID
    int thr_num;                   // thread number (0..)
    int *ws_ptr;                   // workspace pointer
    int *ws_arg;                   // arguments pointer
    void (*func)(int *);           // function to execute
    char *raddr;                   // return address in occam
    int terminated;                // terminated ?
    int cancel;                    // cancelled ?
    int adjustment;                // adjustment for parameters
    sigjmp_buf *jbuf;              // jump buffer
    void *user_ptr;                // pointer to 'spare' space
    void (*cleanup)(void *);       // cleanup function
};
```

When KRoC starts up it creates a small pool of clones, currently 3. This appears to be a sensible default, as the occam web-server (section 5) only needed this many under a sensible load. Figure 1 shows the startup sequence for two clones.

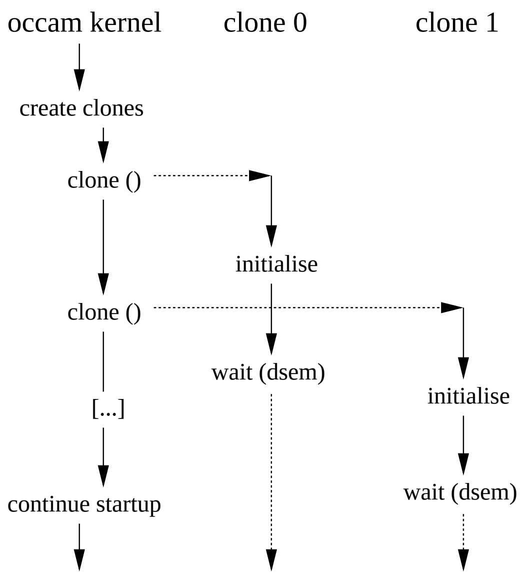

Figure 1: Clone startup

The clones start by entering the function '`clone_entry`', which gets passed the address of this structure as a parameter. The parent process sets '`thr_num`', '`adjustment`' and '`jbuf`' before setting the clone off. It also stores a pointer to the clone's `bsc_thread` structure in the '`clone_arry`' array. When the clone starts executing, it puts its process ID in the '`pid`' field, and in the '`clone_pids`' array. Once the clone has initialised, it blocks on a kernel semaphore claim with all the other non-active clones.

3.3 Dispatching calls

When an occam program calls an external '`B.xxx`' procedure, the actual call is mangled in much the same way as external C calls. This involves the use of some separate glue-code, namely '`call_B_interf`'. This is a simple assembler routine that jumps into the run-time kernel (CCSP), in much the same way that kernel calls (`in`, `out`, `outbyte`, etc.) do. This glue-code handles the workspace much as a normal occam `PROC` would. The run-time kernel expects two arguments from the occam world — the address of the function to call, and a pointer to the arguments. The kernel calls the dispatching function (`bsyscalls_dispatch`), then schedules the next occam process.

The dispatching function is where the information relevant to performing a blocking call is passed to a clone. The following list summarises the jobs performed by the occam-kernel in the `bsyscalls_dispatch` function:

1. The number of available clones is checked for one of two special cases:
   (a) There are no clones left. In this case, a new clone is created immediately and a flag '`deferred_new`' is set to 1. This flag indicates whether or not a new clone should be created before the function returns.
   (b) There is one clone left. In this case, the '`deferred_new`' flag is set to 1, so that there will definitely be a spare clone if and when the next dispatch occurs.
2. '`spl_lock_or_fail (&dispatch_lock)`' is called to lock '`dispatch_lock`'. If the lock cannot be obtained immediately, it is because a previously dispatched job has not been collected by a clone. In such a case '`sched_yield()`' is called to yield the processor in the hope that the clone will be scheduled and the job collected. This will loop until the lock becomes available.
3. Information about the job is placed in the '`dispatching`' structure. This is a structure shared by all, and is the way in which the clones receive information about new jobs.
4. The semaphore on which idle clones wait is notified. This will cause one of the blocked clones to return from its semaphore claim operation.
5. If the '`deferred_new`' flag was set, a new clone is created. This ensures that if the last clone was used this time, then there will be a spare one next time.

From the clone's point of view, things are slightly simpler. The clone will be blocked in an OS kernel semaphore claim (wrapped up by '`claim_semaphore`'). When this function returns, either the semaphore was claimed, or the occam-kernel exited. In the latter case, the clone also exits if the occam-kernel had not already killed it (which it does do on exit).

The clone, after waking-up, simply copies the relevant fields of '`dispatching`' into its own '`bsc_thread`' structure, unlocks '`dispatch_lock`', then executes the requested function. The actual business of executing the call is slightly complicated, as the blocking call can be terminated while in progress. This is discussed in more detail in section 3.5.

Figure 2 shows the main actions of both the occam-kernel and the receiving clone when dispatching a call.

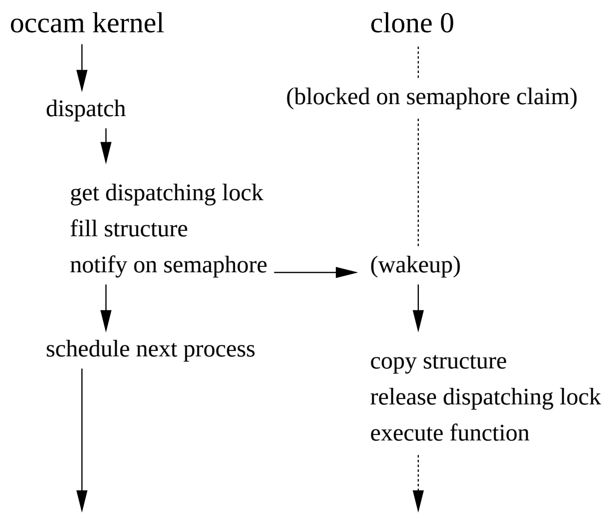

Figure 2: Dispatching a clone

3.4 Collecting finished calls

When the clone finishes, it needs to re-schedule the occam process that dispatched it. This is done by means of a shared queue (`q_fptr`, `q_bptr`), a spin-lock (`queue_lock`) and, if necessary, a signal to the occam-kernel. The clone acquires the spin-lock, adds the process to the queue, increments `num_queued`, then releases the lock. If `num_queued` was previously 0, the occam-kernel process is signalled to collect process(es). If there was already something on the queue before adding this one, it means that the occam-kernel has already been signalled, so should not be signalled again. This is safe since the occam-kernel will not be able to collect the finished calls while the clone holds the lock.

The signal handler on the occam-kernel side sets one of the synchronisation flags, similar to that set by the keyboard or timer. If the occam-kernel was in `safe_pause` (where it idles), it will be awakened [1]. If not, it will pick up the change in the synchronisation flag on the next reschedule. When the next reschedule happens, the occam-kernel locks `queue_lock`, moves the processes to the run-queue, adjusts the number of spare clones, then unlocks `queue_lock`. Figure 3 shows this.

3.5 Terminating (`ALT`*ing*) blocking calls

It is quite possible that a blocking call in the Linux kernel could block forever, or that a function executing aside the occam-kernel could get stuck in a loop. For these reasons, a safe mechanism for terminating them is provided. The implementation of terminating calls is split into two halves. Section 3.5.1 describes how the occam-kernel goes about terminating a call, and section 3.5.2 describes how the clone deals with being terminated. The ability to terminate a blocking call allows, for example, a socket write included as an `ALT` guard. The occam must, of course, be programmed correctly to avoid deadlock. An example of safe termination is given in section 3.5.3.

As mentioned earlier, terminatable blocking calls get passed an extra channel parameter as the first argument. This is internal to the termination mechanism so it is not included in the

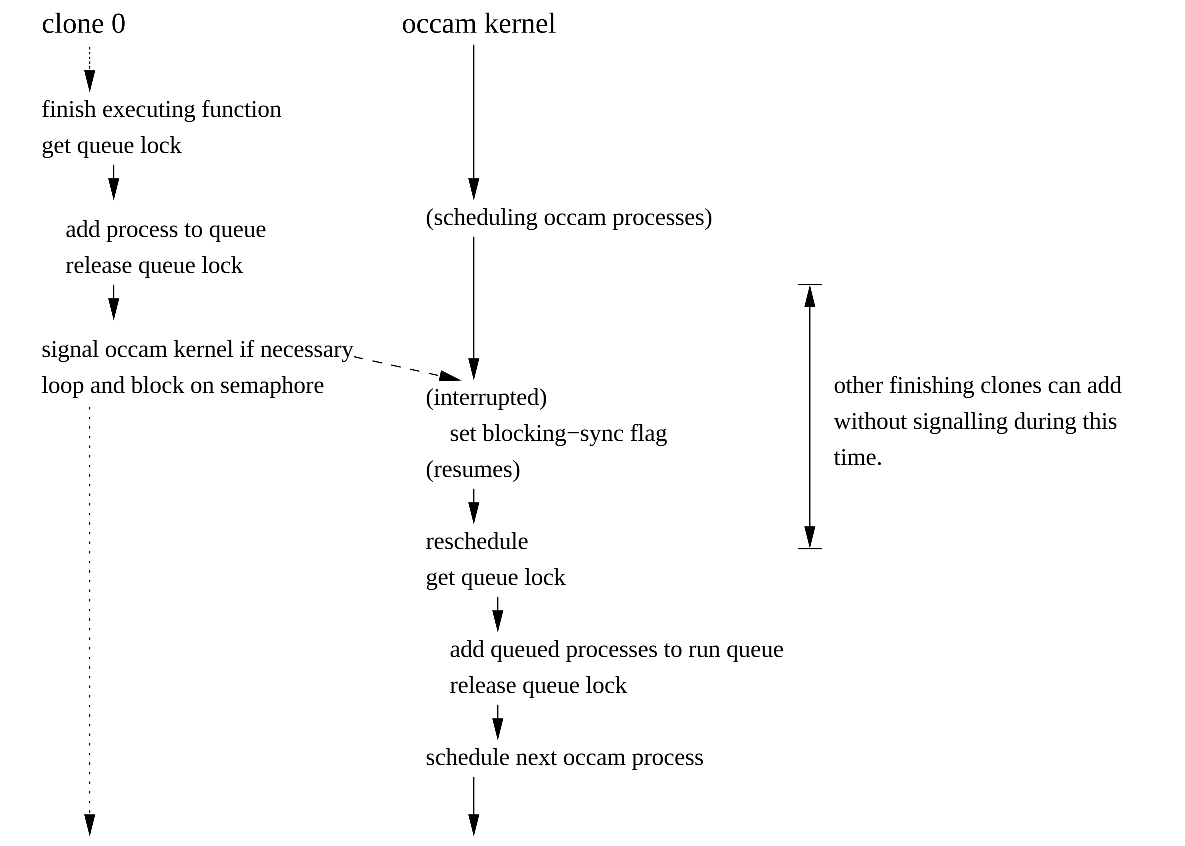
clone 0
occam kernel
finish executing function
get queue lock
add process to queue
release queue lock
(scheduling occam processes)
signal occam kernel if necessary
loop and block on semaphore
(interrupted)
set blocking−sync flag
(resumes)
reschedule
get queue lock
add queued processes to run queue
release queue lock
schedule next occam process
other finishing clones can add
without signalling during this
time.

Figure 3: Collecting a finished call

arguments given to the C function being called. The '`adjustment`' element of the clone's `bsc_thread` structure is used here to shift the parameters passed one to the left. Before the clone starts executing the call, it places a pointer to its own `bsc_thread` structure in the channel word. As this is most certainly *not* an occam workspace pointer, the occam program must not attempt direct communication on it. The '`cancel`' and '`terminated`' members of the `bsc_thread` structure are used to provide race-free termination, initially being set to zero.

3.5.1 Initiating termination from occam

Once a blocking call (or other computing function) is in progress, an occam process running in parallel with it is able to terminate it. This is done through the use of a built-in function declared to the occam world as:

```
#PRAGMA EXTERNAL "PROC C.killcall (CHAN OF INT c, INT result) = 0"
```

An occam process can call this function, passing the same channel parameter as was passed to the blocking call. The second parameter '`INT result`' is used to return the result of the termination, with the following meanings:

-1	the blocking call had already finished
0	the blocking call was successfully terminated
1	the blocking call had only just finished
2	the blocking call is currently finishing

The most common case will be the second (terminated successfully). The other three cases (-1, 1 and 2) indicate various grades of non-successful termination. The following algorithm implements the occam-kernel side of the termination:

```
1:  $T \Leftarrow$ contents of channel word
2:  **if** $T$ = null **then**
3:     //call has either not started or has finished
4:     sched_yield()
5:     $T \Leftarrow$ contents of channel word
6:     **if** $T$ = null **then**
7:        //call has finished
8:        **return** -1
9:     **end if**
10: **end if**
11: **if** $T$[terminated] **then**
12:    //call has finished, but has not yet cleared channel word
13:    **return** 1
14: **end if**
15: $F \Leftarrow 1$
16: //atomically swap contents of T[cancel] with $F$
17: **swap** $T$[cancel], $F$
18: **if** $F = 1$ **then**
19:    //call is in the process of terminating
20:    **return** 2
21: **end if**
22: //send a SIGUSR1 signal to the clone's OS process
23: **signal** $T$[pid], SIGUSR1
24: **return** 0
```

The atomic swap at line 17 indicates the point where the occam-kernel either commits to terminating the clone (via a signal), or leaves it alone (because it is in the process of terminating itself). On a uni-processor machine, it is entirely possible that the occam program attempts termination before the clone has scheduled. In this case, '`sched_yield()`' (line 4) is called to reschedule the occam-kernel OS process. This gives the clone the chance to execute the call before it is killed, and it is only killed if it does not complete before the occam-kernel gets rescheduled. In some situations, this might not be the desired behaviour – we actually might want to terminate the call before it starts. It is unlikely however, as the signalling on the semaphore (to dispatch a clone) will cause the Linux kernel to favour the first process waiting on that semaphore, in this case, the clone which being dispatched.

3.5.2 Dealing with termination in the clone

All of the clones install a signal handler for the SIGUSR1 signal, which is used to 'interrupt' them. This signal handler is used in conjunction with a `sigsetjmp`/`siglongjmp` pair, so that execution can continue at a well-known place when the signal is delivered. A call to `sigsetjmp` is made by the clone just before the function is executed. The '`jbuf`' member in the clone's `bsc_thread` structure points at the jump buffer, which is located just beyond the `bsc_thread` structure in the clone's stack. The `sigsetjmp` function returns 0 if it is returning directly, or 1 if it is coming back from a `siglongjmp`, in our case from the signal handler.

The following algorithm implements the clone's half of the termination process, starting before the blocking call is made:

```
1:  $T \Leftarrow$ pointer to the clone's bsc_thread structure
2:  $T$[cancel] $\Leftarrow 0$
3:  $T$[terminated] $\Leftarrow 0$
4:  channel word $\Leftarrow T$
5:  **if sigsetjmp** $T$[jbuf] $= 0$ **then**
6:      //*returning directly*
7:      **unblock** SIGUSR1
8:      make blocking system call
9:      $T$[terminated] $\Leftarrow 1$
10:     **block** SIGUSR1
11:     $S \Leftarrow 0$
12: **else**
13:     //*returning from the SIGUSR1 signal handler*
14:     $T$[terminated] $\Leftarrow 1$
15:     $S \Leftarrow 1$
16: **end if**
17: $F \Leftarrow 1$
18: //*atomically swap contents of T[cancel] with $F$*
19: **swap** $T$[cancel], $F$
20: **if** $F = 1$ **and** $S = 0$ **then**
21:     //*other side committed to terminating, but signal is pending*
22:     **wait** SIGUSR1
23: **end if**
24: channel word $\Leftarrow$ null
```

The functions **block** and **unblock**, block and unblock a signal respectively. If a signal is delivered while it is blocked, it is marked as pending, and delivered when it is unblocked or **wait**ed for. There is a very unlikely case where the occam-kernel terminates a call, just as the call itself is terminating, but where the signal has not been delivered. In this case SIGUSR1 is waited for, and when delivered, execution resumes at the point where `sigsetjmp` was called.

Figure 4 shows the actions of both parties during a 'normal' termination.

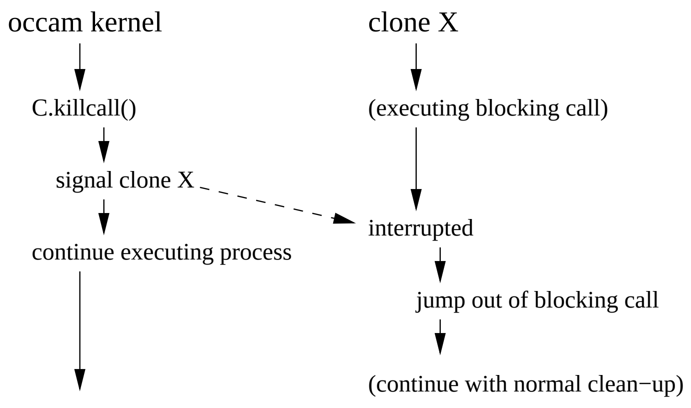
occam kernel
clone X
C.killcall()
(executing blocking call)
signal clone X
interrupted
continue executing process
jump out of blocking call
(continue with normal clean−up)

Figure 4: Normal termination of a blocking call

3.5.3 Safe termination

The occam fragment below shows a method for terminating a blocking call, without causing deadlock:

```
INT kill.result:
BOOL did.kill:
SEQ
  CHAN OF INT c:
  CHAN OF BOOL signal:
  PAR
    --{{{  blocking call
    SEQ
      BX.whatever (c, ...)
      signal ! TRUE
    --}}}
    --{{{  collection/termination
    PRI ALT
      BOOL any:
      signal ? any
        did.kill := FALSE            -- normal finish
      ... termination condition
        SEQ
          C.killcall (c, kill.result)
          BOOL any:
          signal ? any
          did.kill := TRUE
    --}}}
```

In this example, the "`... termination condition`" could either be an input guard, or a timer guard. At the end of the code fragment, '`did.kill`' indicates whether or not the blocking call was terminated, and if so, '`kill.result`' indicates the outcome of the termination. The KRoC occam socket library [6] uses this technique to provide `ALT`able variants of the `read`, `write`, `accept` and `recvfrom` `PROC`s through a two-channel interface. This reduces the application's involvement to:

```
CHAN OF BOOL kill:
CHAN OF INT response:
PAR
  --{{{  blocking call
  socket.altable.X (kill, response, ...)
  --}}}
  --{{{  wait for response or timeout
  TIMER tim:
  INT t:
  SEQ
    tim ? t
    PRI ALT
      --{{{  incoming response for normal termination
      INT any:
      response ? any
        kill ! TRUE
      --}}}
      --{{{  timeout (after 1 second)
      tim ? AFTER (t PLUS 1000000)
        INT k.result:
        SEQ
          kill ! TRUE
          response ? k.result
          ... take action on k.result if necessary
      --}}}
  --}}}
```

This fragment demonstrates a killable blocking socket operation (‘socket.altable.X’) using the two channels ‘kill’ and ‘response’. The design rule is that whatever happens, a single communication must occur on each channel. This ensures that ‘socket.altable.X’ will not deadlock, as it too must communicate once on each of these channels. In the case where the call terminates of its own will, ‘socket.altable.X’ performs a PARallel input/output, allowing the user to order the communications in whatever way is appropriate to the application.

3.5.4 Cleaning up after termination

In some cases, the C function being executed may need to perform some cleaning-up operations after termination from the occam world. The bsc_thread structure provides two additional members to aid this operation. The first, ‘user_ptr’, is a pointer into the clone’s stack, just above the bsc_thread and jump-buffer structures. The second, ‘cleanup’ is a pointer to a function which will be called if the call is terminated. To manage this from the C world, the following C function is provided:

```
extern void *bsyscalls_set_cleanup (void (*)(void *));
```

If the blocking call needs this functionality, it should call this, passing a pointer to a separate clean-up function. The ‘user_ptr’ pointer is returned as the result. When the cleanup function is called, it is given ‘user_ptr’ as an argument. If a blocking call needs to clean up after termination, it will also probably need some of its previous state, which should be stored at this pointer. The space available starting at ‘user_ptr’ is determined largely by the clone’s stack size (currently 128 kilobytes), minus the amount of stack the clone is currently using. It is recommended that no more than 1 page (4096 bytes) be used.

One perceived, and investigated, use of this is when a blocking call launches another OS process, and waits for it to finish. If this functionality were not used, the OS process started would not be killed when the blocking call is terminated. With this functionality, the blocking call could arrange for the OS process to be killed on its own termination.

4 Performance

The performance of blocking system calls is assessed largely by how much impact the system takes when dispatching, handling and collecting the calls. This in turn provides an indication of how fine-grained the parallel usage of them may be. Figure 5 shows the arrangement of the machines involved on the network.

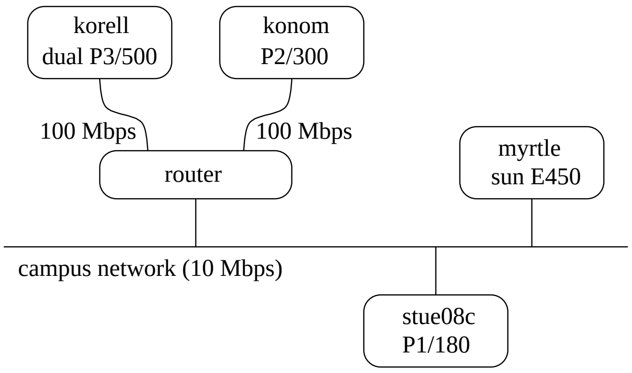

Figure 5: Test network configuration

Figure 7 shows the impact on computation from communication, with the corresponding throughputs in figure 8. The benchmark used was similar to the one described in [7], but modified to use streaming TCP data, as shown in figure 6. Data was streamed from the machines `konom` and `stue08c` to `korell`, once with `korell` using a single processor to handle the blocking calls, and again with `korell` using both processors. Communication between `konom` and `korell` happened at 100 Mbps, with a routing switch forwarding the packets in cut-through mode (a form of worm-hole routing), whereas communication between `stue08c` and `korell` happened at 10 Mbps. In this latter case, the network latency is much higher, as the "campus network" includes a 100 Mbps FDDI ring and several routers between the machines.

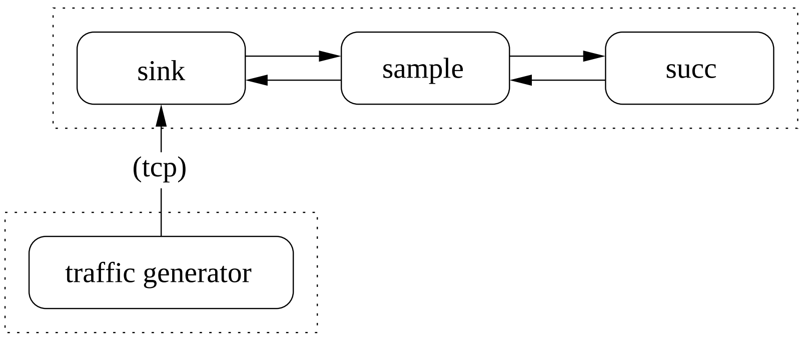

Figure 6: Benchmark process network

For runs between `stue08c` and `korell`, the difference between single and dual processor performance is marginal. As would be expected, the dual processor throughput in this case is slightly better than the single processor, up to the point where packet fragmentation occurs (around 1500 bytes). Runs between `konom` and `korell` tell a different story, with the results for single and dual processor performance being somewhat different. In the case of `korell`

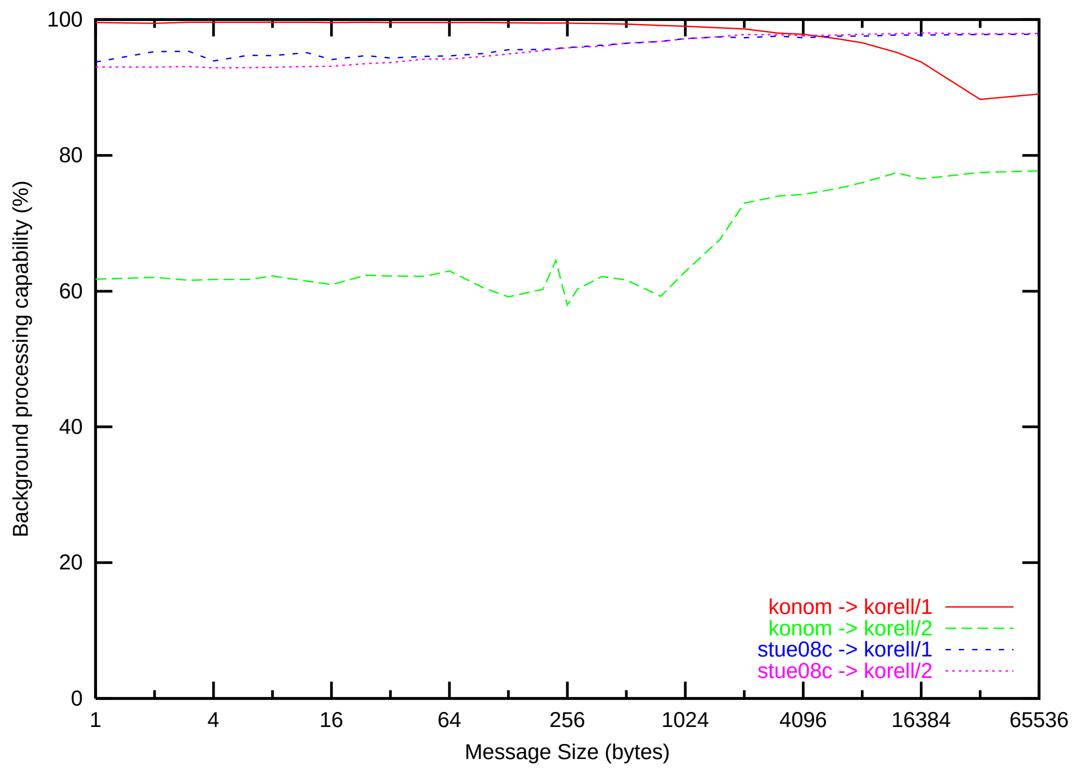

Figure 7: Impact of communication on computation

having both processors available to schedule the blocking calls, throughput is high, causing a notable hit in background processing capability. In the case where only one of `korell`'s processors is available, throughput is dramatically reduced. Here, the throughput is less than that of the 10 Mbps network, up to a packet size of around 9000 bytes, where the limit of the 10 Mbps network is reached. One cause of this poor performance stems from the low latency between the two machines. As the occam-kernel and the blocking calls are restricted to a single processor, a dispatched clone must wait for a (Linux) kernel reschedule before becoming active. At the point where the clone executes the `read()` system call, the sender is likely to have stalled, as the acknowledgement packets will come back much faster than in the case of the 10 Mbps network. The difference in throughput on the 100 Mbps network is around a factor of 250, and this 10 Mbps single processor case is the only one of the four runs which fails to reach maximum theoretical throughput. When running at 100 Mbps with both processors available, blocking call dispatch times are much lower than the single processor times. This is expected, as the Linux kernel will schedule the blocking call immediately if the second processor is inactive.

To provide additional performance metrics, the occam-kernel can be compiled in such a way that it profiles blocking calls. The pentium CPU cycle counter is used to get accurate time-stamps for *dispatch*, *trigger*, *kill* and *finish* events. The *dispatch* time-stamp is set when the occam-kernel dispatches the call to the clone, and the *trigger* time-stamp is set by the clone when it picks up the call. If the call is terminated, the *kill* time-stamp is set at the point where the occam-kernel initiates termination (in `C.killcall()`). When the call finishes, the clone sets the *finish* time-stamp. It should be noted that profiling the blocking calls decreases their performance, as the data is dumped directly to the standard error stream.

Figures 9 and 10 show the behaviour of the clones during benchmarking of `occwserv` (section 5). Each vertical 'strip' represents one of the clones, and in both cases there are 12.

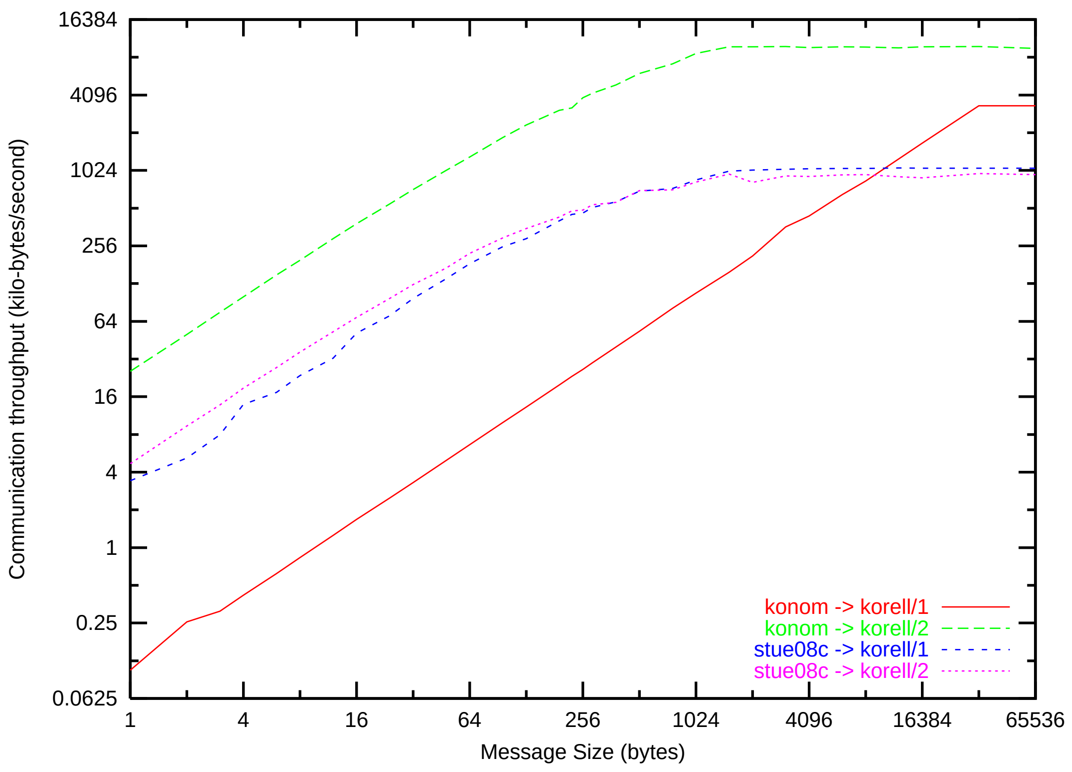

Figure 8: Data throughput rates during communication

At each point where a clone is dispatched from the occam-kernel, a horizontal line is drawn from the left-hand margin to the clone. The shaded regions represent a blocked clone (more accurately, a clone executing the requested function).

Figure 9 shows the behaviour when `occwserv` was benchmarked from `myrtle` (figure 5) using the apache benchmark program `ab`. This varies greatly from the trace obtained when `occwserv` was benchmarking from `stue08c` using the occam benchmark program `ob`.

`ab` benchmarks by making n requests to the server, then waits for all the responses before setting off the next n requests. Figure 9 shows this quite clearly. The clone which blocks until the next 'barrier' is the acceptor, as it will block until `ab` dispatches the next set of calls. This call does not actually make it through the 'barrier' as the trace might suggest. The occam benchmark program on the other hand attempts to maintain n requests at all times, by making a new request as soon as an existing one finishes. The difference can be clearly seen in figure 10, where dispatching is distributed more evenly than in figure 9. Both of the traces were made when `occwserv` was using both processors, in an effort to minimise the impact of profiling.

The following table shows the results of profiling `occwserv`, being benchmarked by `ob`, for both single and dual processor instances of `occwserv`. This quantifies the difference between executing blocking system calls on a uni-processor, and executing them on a dual-processor.

	Measured	**Min (us)**	**Avg (us)**	**Max (us)**
`occwserv/1`	*dispatch* to *trigger*	3.28	79.51	2,356.87
$\rightarrow$`ob`	*trigger* to *finish*	46.92	248,792.19	3,292,883.81
`occwserv/2`	*dispatch* to *trigger*	3.96	41.14	1,158.79
$\rightarrow$`ob`	*trigger* to *finish*	17.83	218,370.73	2,745,141.51

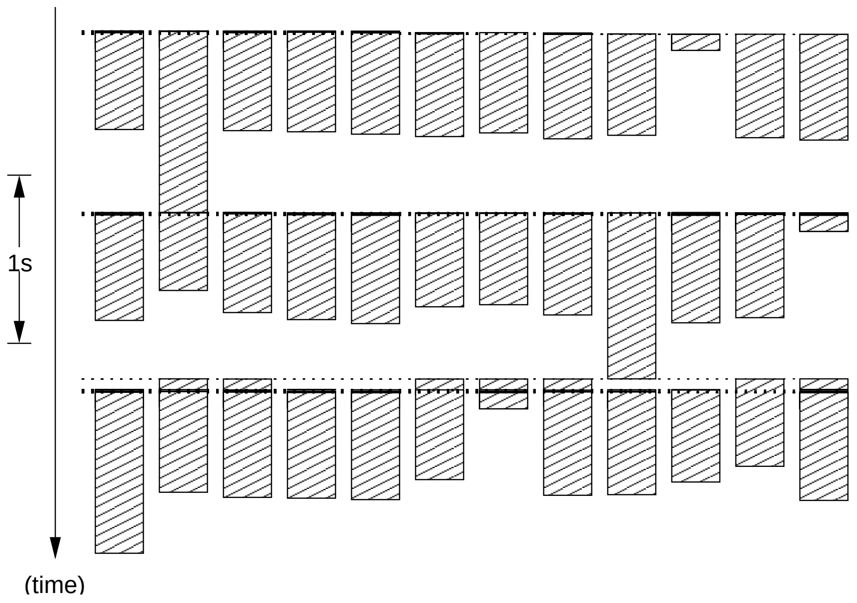

Figure 9: Execution profile of blocking system calls with `ab` (apache benchmark) showing the ‘active’ time of each clone

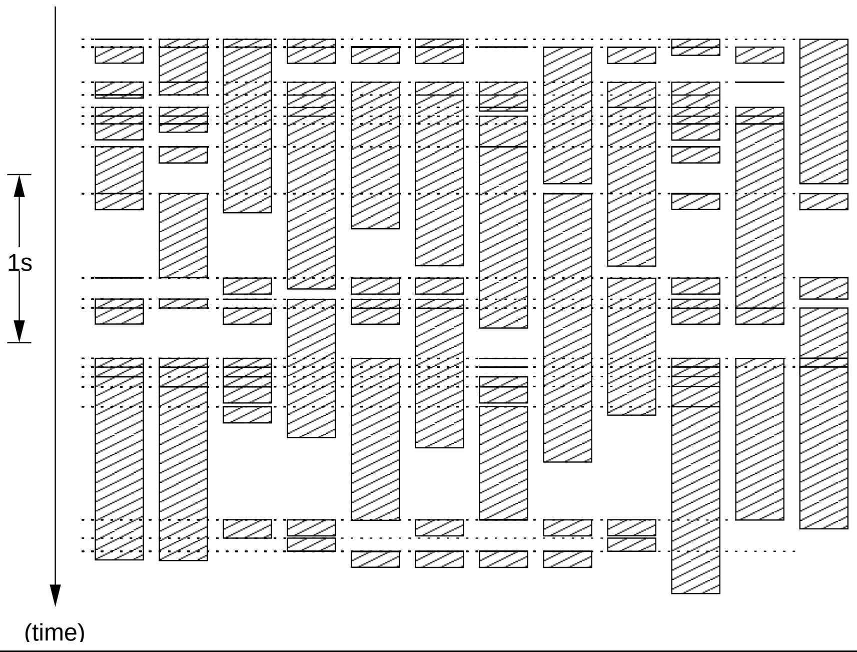

Figure 10: Execution profile of blocking system calls with `ob` (occam benchmark) showing the ‘active’ time of each clone

5 An occam Web-Server

This section describes `occwserv` – the occam web-server. The overall organisation of the server is shown in figure 11. The vertical dots indicate process replication. Currently there are 128 '`read.line`→`switch`' processes, 256 '`file.get`' processes, and 32 '`cgi.get`' processes. `occwserv` uses the occam socket, file and process libraries, as described in [6], to handle interaction with the OS.

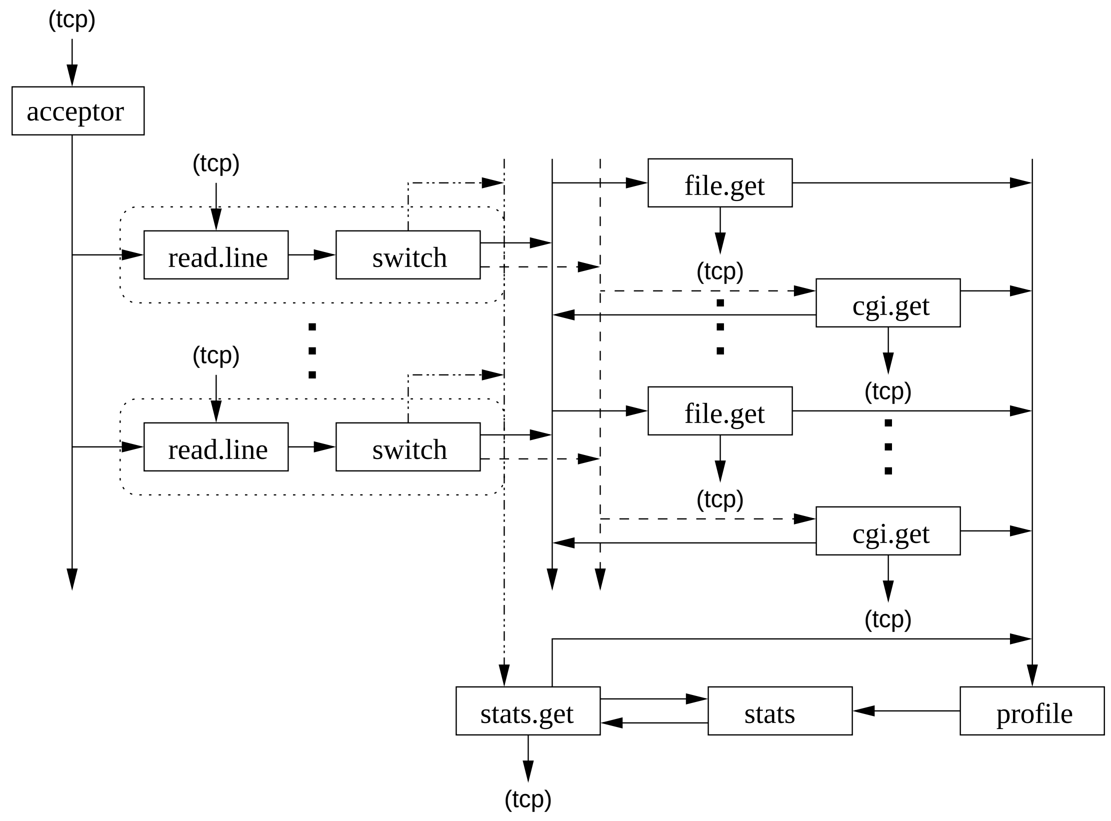

Figure 11: Network diagram for the occam web-server

After initialisation, all processes will be blocked on either channels or occam3[8] style shared-channels (implemented via the semaphore abstract type [9]), with the exception of the '`acceptor`' process, which will be blocked in the Linux kernel on a socket accept. The '`acceptor`' process consists of a loop, which accepts an incoming connection, time-stamps it, then passes it on to one of the '`read.line`' processes through a *one-to-any* channel. The server is constructed such that each '`read.line`' has an associated '`switch`' process, to which it is connected. These two processes perform the bulk of data processing in the server.

The function of the '`read.line`' process is to read data from the client connection, separate it into lines, and pass each line to the '`switch`' process. After a blank line has been read, or after an error has occurred, the client connection is passed to the '`switch`' process and '`read.line`' loops to wait for another client. The '`switch`' process examines the incoming data, presumably an HTTP request, and decides where to send it based on that request. If the input is invalid (non-HTTP), the connection is sent to one of the '`file.get`' processes, with a request to produce an error file.

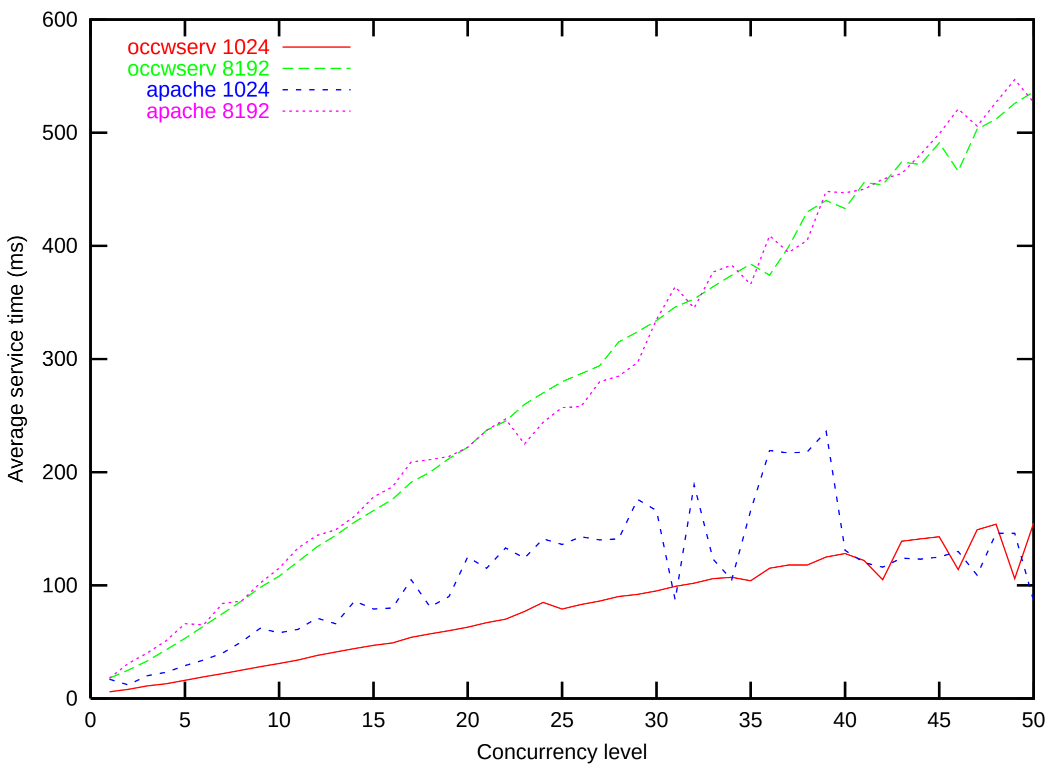

Figure 12: Average time to service a request (standard apache)

The collection of processes just described form the 'front-half' of the web-server. The rest of the network deals with returning data to the clients and collecting statistics about the connections. Requests for the special page "`stats.html`" are sent directly to the '`stats.get`' process, through an *any-to-one* channel. '`stats.get`' interacts with the '`stats`' process to produce a page containing information about connections and the server in general. Once the data has been sent back to the client, the client connection is passed to '`profile`', along with other client connections from the '`file.get`' and '`cgi.get`' processes.

As mentioned earlier, each client connection is time-stamped as it is accepted. Upon receipt, '`stats.get`', '`file.get`' and '`cgi.get`' add a second time-stamp before processing. Once the connection has been processed, it is passed along to '`profile`' through an *any-to-one* channel. '`profile`' takes the interval between these two times, along with other information, and passes it to the '`stats`' process, which keeps running totals.

The '`file.get`' and '`cgi.get`' processes are connected through two *any-to-any* channels respectively. '`file.get`' dumps the requested file to the client, or dumps an error file if it cannot. Once the output has been written to the client, the connection is passed to '`profile`'. The data is copied using the `file.fd.fd.copy` procedure, as it keeps the data copying within a single clone, returning upon completion or error. '`cgi.get`' executes the requested script, with the socket file-descriptors connected directly to the output of the script, thus avoiding any data copying in `occwserv`. If an error occurs while processing the script (script does not exist, script dumped core, etc.), then the connection is fed back into '`file.get`' farm to produce an error for the client. This saves some code duplication by re-using the services of '`file.get`' to handle errors for '`cgi.get`', and anything else if it were added.

If the client connection is thought of as the endpoint of a channel, then `occwserv` effectively passes that endpoint around the network, demonstrating a use of the channels over channels idea [10].

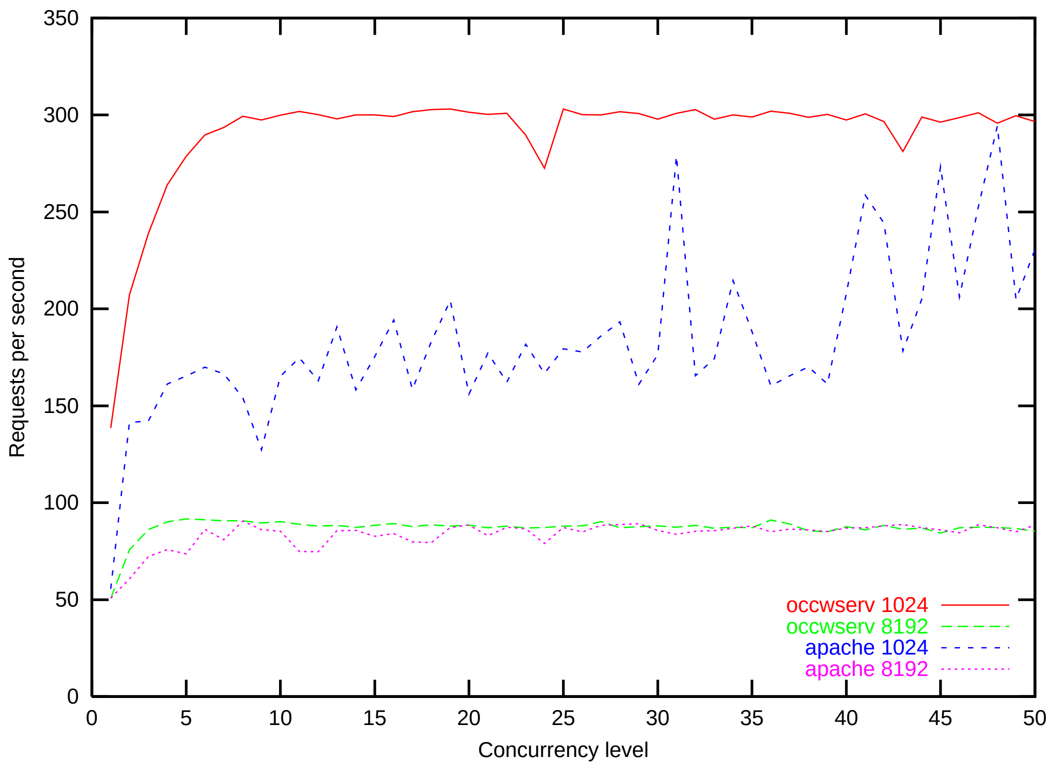

Figure 13: Number of requests handled per second (standard apache)

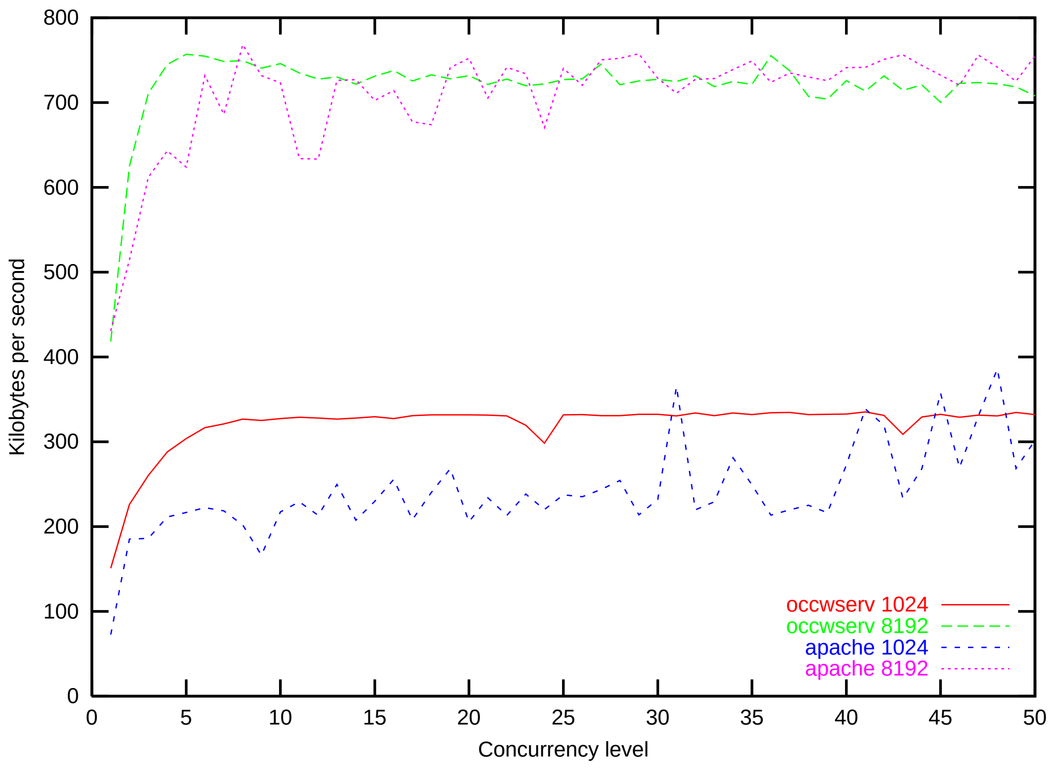

Figure 14: Data throughput rate (standard apache)

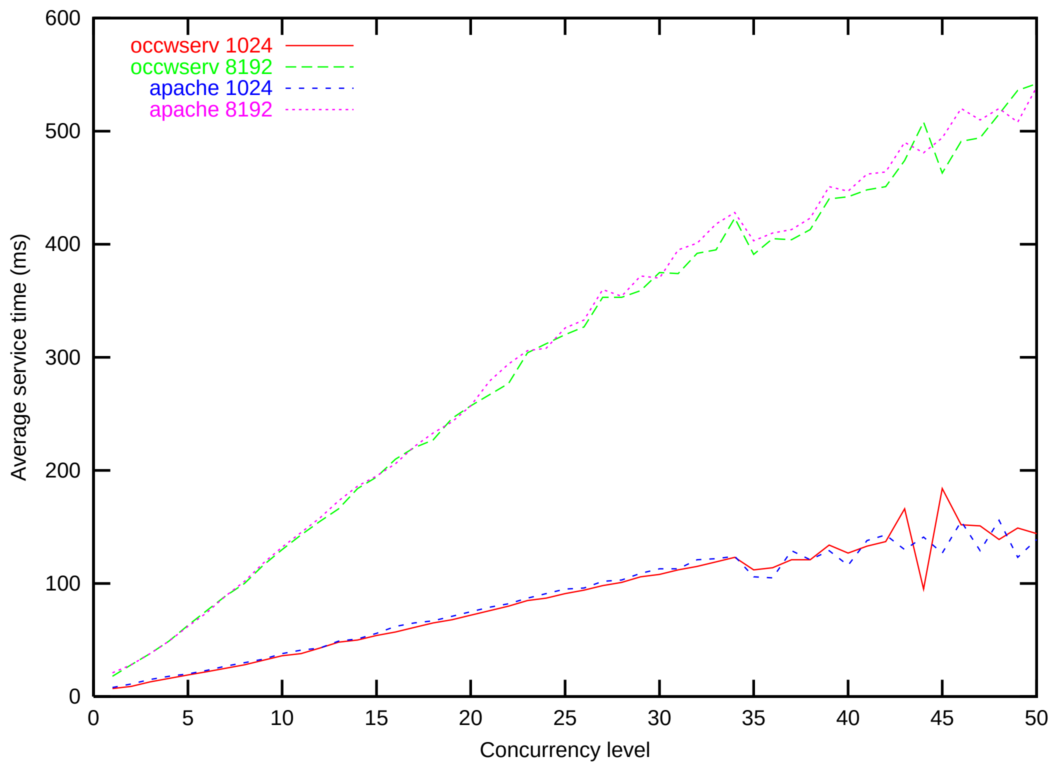

Figure 15: Average time to service a request (tweaked apache)

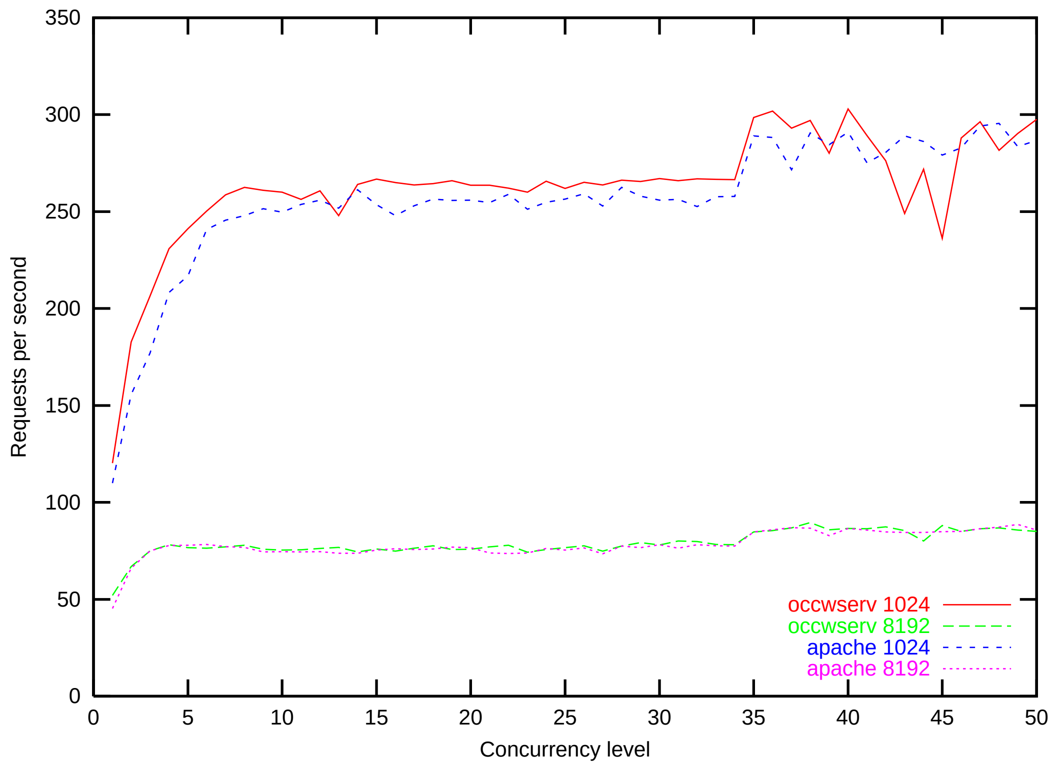

Figure 16: Number of requests handled per second (tweaked apache)

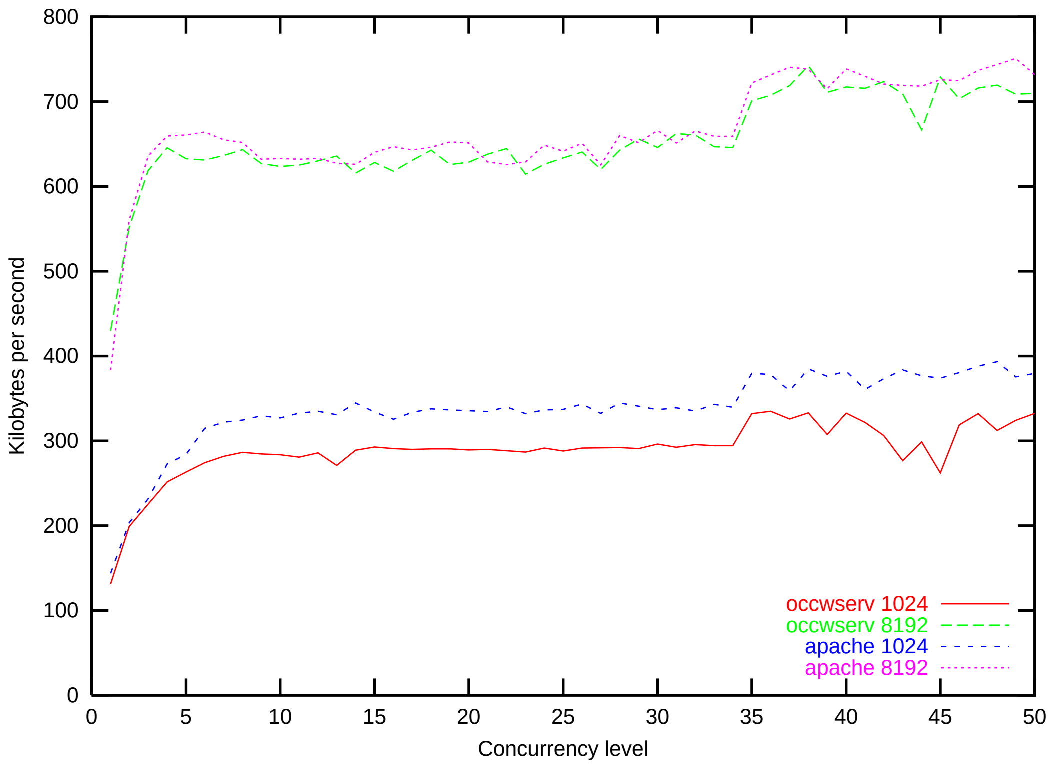

Figure 17: Data throughput rate (tweaked apache)

5.1 *Performance of* `occwserv`

The apache benchmark program, `ab`, was used to benchmark `occwserv`. This program comes as part of the standard apache web-server source distribution. `ab` takes a URL, number of requests and concurrency level as parameters, hitting the URL specified the requested number of times, and tries to ensure that ‘concurrency level’ requests are processed simultaneously. This parallelism is achieved through using the ‘`readv()`’ system call, not through a scheduling mechanism. As the benchmarking was performed over a real network, real network factors will bias the results, but hopefully putting (on average) an equal bias one each of `occwserv` and apache. The physical network connectivity can be seen in figure 5, where `occwserv` and apache were run on korell, and `ab` was run on stuE08C.

Figure 12 shows the average service times for `occwserv` and apache at varying levels of concurrency (1 to 50). One pair of lines shows the performance when retrieving a 1 kilo-byte file, and the other pair when retrieving an 8 kilo-byte file. When retrieving 1k files, `occwserv`’s average service time is on the whole faster than apache. With 8k files, `occwserv` performed faster up to a concurrency level of about 22, at which point the service times become comparable with apache.

Figure 13 shows the number of requests handled per second, at varying levels of concurrency, for 1k and 8k files. Surprisingly, `occwserv` dominates this somewhat, and in the case of 8k files, this value is almost constant from a concurrency level of 5 onwards. The likely causes for this are apache’s additional client processing, which `occwserv` does not perform, and apache’s habit of killing its server processes (described below).

Figure 14 shows the corresponding data throughput rates from `occwserv` and apache, to the benchmark program `ab`. As can be seen, `occwserv` delivers a higher throughput than apache when serving 1k static pages. For 8k static pages, `occwserv` is initially faster, but apache quickly catches up, becoming comparable around a concurrency level of 13.

As can be seen from these three graphs, apache behaves rather erratically when compared with the smoother behaviour of `occwserv`. This behaviour arises from the way apache handles it's server processes. A configuration option specifies the number of connections a server process may respond to before it gets restarted. This is done to stop the server getting thrashed, and to reduce the impact of any memory leaks. The default setting in apache is 30 requests per server process. Figures 15, 16 and 17 show the results of the same tests when this limit in apache was increased. The limit was set to one million to completely avoid the restarting of server-processes during benchmarking. These additional results show that apache and `occwserv` are mostly comparable in terms of performance. Apache was able to transfer 1k files faster than `occwserv` (figure 17), but `occwserv` handled more requests for 1k files per second (figure 16). For 8k files, apache and `occwserv` have a roughly equal performance.

6 Conclusions and Future Work

This paper has shown that it is possible to enhance a user-level thread scheduler (CCSP [5]) such that individual processes may block inside the OS kernel, without stopping other user-level threads (occam processes) running in parallel. The work done here centers around the KRoC occam system, but it could be ported to other user-level thread schedulers such as MESH [11], which was based initially on CCSP.

The simplicity of the occam web-server demonstrates that occam is a natural language when it comes to programming multi-threaded internet applications. In addition to the ease and low-cost[2] of fine-grained parallelism from within occam, required for applications such as web-servers, the occam compiler will perform parallel usage checking to ensure parallel safety. Low-cost in occam's terms also means lower development, maintanence and enhancement costs. The usage checking performed within the occam compiler reduces the number of potential bugs by ensuring that parallel processes adhere to certain design rules. The web-server is currently 1500 lines of occam code and took less than a week to develop, from design to production. The overall design was, for the most part, intuative, and `occwserv` could easily be extended by "plugging in" more functionality. It should also be relatively easy to call on external programs to perform server-side processing, such as PHP [12], by using a strategy similar to the CGI handling processes.

The general technique of handling blocking system calls in the way need not be tied to Linux at all. However, one of the features which this technique uses quite heavily is the ability to share memory and file-system context between different OS processes. This removes the problem of data copying between occam and the C world when a blocking call is made. Most major UNIX variants include the ability to share memory between threads and/or processes, usually through the use of `mmap()` or System-V style shared-memory (SHM). The structures used to pass information to and from the clones, and the occam workspace could be placed in this shared region, yielding the same functionality as we have implemented. Two state-of-the-art Real-Time Operating Systems (RTOSs), Lynx-OS and QNX also have a notion of

[2]153ns context switch on an Intel P3 500Mhz machine

threads, so this technique could be applied to them also. It is envisaged that operating-systems on which this technique could be used would provide sufficient synchronisation primatives to control the threads.

Sharing the file-system context is necessary to allow a file-descriptor opened in one clone to be used in another clone. In systems without explicit file-system context sharing, the descriptors can be passed between the processes using a pipe – this technique is non-trivial however. If a clone opens a descriptor, then passes it to another clone, both clones must `close()` the descriptor to dispose of it. One solution to this would be to provide a mapping of file-descriptors to clones, forcing blocking calls involving a particular descriptor to be executed by the clone which holds that descriptor. The mapping would need careful construction, as two (or more) clones could refer to different files by the same numeric file-descriptor value.

With some kernel modifications, it would be possible to remove the need for the clones, and let the Linux kernel handle the blocking of user-level threads. This of course would require the kernel to know explitictly about the threads, and probably require the kernel to schedule them as well. This would mean that each user would have to patch their Linux kernel – which may not be convienient. The potential performance benifits of this approach are quite attractive however, as the Intel Pentium 2 processor, and above, provide speedy instructions to enter and leave the OS kernel (`sysenter` and `sysexit`) [13]. The occam processes would execute in user-space, much as they do now, but would call on CSP [14] functionality in the Linux kernel to handle communication, synchronisation and parallelism. Such an approach also eliminates many implementation difficulties, such as idling (currently with `safe_pause()`[1]), I/O, and multi-processor management.

Acknowledgements

I would like to express my thanks to Peter Welch for providing ideas and supervision during the construction of this paper, to Jim Moores, David Wood and Michael Poole for explaining various aspects of the transputer, KRoC and `tranpc`, and to Jeff Cordova who pointed me to the feature in apache which caused the erratic behaviour as shown in figures 12, 13 and 14.

References

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

[2] SGS-Thompson Microelectronics Limited. *T9000 occam 2 Toolset Handbook*. SGS-Thompson Microelectronics Limited, 1994. Document number: 72 TDS 457 00.

[3] 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*, volume 52 of *Concurrent Systems Engineering*, Address, April 1998. WoTUG, IOS Press.

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

[5] 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.

[6] Fred Barnes. *Socket, File and Process Libraries for occam*. Computing Laboratory, University of Kent at Canterbury, June 2000. Available at: `http://www.cs.ukc.ac.uk/people/rpg/frmb2/documents/`.

[7] Peter H. Welch and Michael D. Poole. occam for Multi-Processor DEC Alphas. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 152–174, University of Twente, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press, Netherlands.

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

[9] 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, University of Twente, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press, Netherlands.

[10] Henk L. Muller and David May. A Simple Protocol to Communicate Channels over Channels. Technical report, University of Bristol, Department of Computer Science, January 1998.

[11] 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, April 1999.

[12] PHP Group. PHP: Hypertext Preprocessor. Available at: `http://www.php.net/`.

[13] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999. Available at: `http://developer.intel.com/design/PentiumIII/manuals/`.

[14] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.