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



# An Experiment with Recursion in occam

David C. WOOD
*Computing Laboratory, University of Kent at Canterbury, CT2 7NF*
D.C.Wood@ukc.ac.uk

**Abstract.** An experimental version of KRoC [1] has been written that implements recursion in occam*, using Brinch Hansen's algorithm for allocating activation records [2]. It shows that efficient parallel recursion is possible in occam.

## 1. Introduction

This paper describes an experimental implementation of recursion in occam. It is based on the SPARC version of KRoC, so it depends on the implementation of routines (`PROC`s and `FUNCTION`s) on the transputer. For simplicity, no changes have been made to the compiler itself, only to `octran`, the translator from transputer to target assembly language. It is not a fully working system, but it demonstrates the potential of the method.

## 2. Why occam does not support recursion

Recursion, dynamic arrays, and parallel constructs with variable numbers of processes are not allowed in occam.

The reason for this is that it is very important, in many applications, that the maximum amount of memory required by a program should be known at compile time, so that it is impossible for an executing program to run out of memory. This is particularly true for programs involving parallelism, where the occurrence of such errors may depend on details of the scheduler.

This limitation has long been a minor embarrassment to occamists, since recursion is such a valuable tool. In particular, parallel recursion is a natural way of expressing many algorithms.

### 2.1. *Syntax*

It is sometimes said that the lack of recursion in occam is merely an implementation restriction, but actually it is impossible to express it in the syntax of the language. A name comes into scope only at the end of its declaration, so any attempt at a recursive call would be a reference to an undeclared routine (or possibly to an earlier routine of the same name – see below). This rule is also important in other sorts of declarations, allowing such idioms as

```
VAL [][]REAL64 Y IS Y:
```

so it cannot be changed.

* occam is a trademark of the SGS-THOMSON Microelectronics Group.

### 2.2. *Pseudo-recursion in* occam

In fact it is possible to implement recursion to a limited depth in occam [3], using this syntactic feature of declarations: the routine is simply written out several times, and each 'recursive' call refers to the immediately preceding copy.

## 3. Adding recursion to occam

The problems here are to find some way of expressing recursion in occam without doing violence to the syntax of the language, and to allocate memory to routines in a way that allows both parallelism and recursion, and is reasonably efficient.

### 3.1. *Activation records*

The *activation record* of a routine is a block of memory that holds all its local variables (including temporaries generated by the compiler), the parameters passed into it, and control information like the return address.

In a block-structured language without parallelism, routines are always entered and left in a last-in–first-out order, so activation records may be kept on a simple stack. They are then usually called *stack frames*. With this mechanism, recursion incurs no extra cost. (Some compilers allow recursion in Fortran, which is not supposed to support it, because preventing it would be more difficult.)

For convenience, a *frame pointer* is often used to point to the current stack frame, since the stack pointer may change during the execution of the routine – for example, when setting up the parameters for a nested call. The stack frame record is then roughly the region between the frame pointer and the stack pointer. In this case, the previous frame pointer may be saved in the stack frame, giving a linked list of activation records.

When one routine calls another, they communicate through parameters. Since these are set up in one routine and used in another, they can be considered to be in a region where the two stack frames overlap.

On the transputer, the *workspace pointer* corresponds to the stack pointer, but there is no frame pointer, so the boundaries between activation records are not so obvious.

### 3.2. *Dynamic allocation of workspace*

The conventional implementation of recursion using a stack does not work when processes are running in parallel, since entries to and returns from routines are no longer simply nested. Hence some other mechanism for allocating memory is required. A general heap system would clearly work, but is likely to be slow.

### 3.3. *Parameters*

If the activation records of the calling and called routines are no longer adjacent, the handling of parameters is more complicated. Ideally, to avoid unnecessary copying, the new record should be allocated by the calling routine, so that the parameters can be stored in it.

On the transputer [4], and hence in KRoC, up to three parameters are placed in the registers of the integer evaluation stack before the call, and the call instruction itself saves them in the workspace, together with the return address. Any further parameters are placed in the workspace before the call.

As well as the explicit parameters, additional 'hidden parameters' may be passed to a routine. These include a pointer to the vector space, used for arrays and records, and static links, for access to non-local variables.

### 3.4. *Brinch Hansen's algorithm*

Brinch Hansen has described a simple and efficient algorithm for allocating memory in parallel recursive programs.

Initially, there is a single block of free memory (`memory` below) from which all activation records will be taken. Each routine has its own 'pool' of free records, which is initially empty.

When a routine is called, its activation record is taken either from its pool, or, if that is empty, from the free memory. On a return, the activation record is released to the pool.

Records are never returned to the free memory, and hence cannot be reused by other routines.

Brinch Hansen gives his algorithm in Pascal, as follows:

```
var
  pool   : array [1 .. limit] of integer;
  memory : array [min .. max] of integer;
  top    : integer;

procedure initialize;

  var
    index : integer;

  begin
    for index := 1 to limit do
      pool[index] := empty;
    top := min - 1              { $$top := min }
  end;

procedure allocate (index, length : integer;
                      var address : integer);

  begin
    address := pool[index];
    if address <> empty then
      pool[index] := memory[address]
    else
      begin
        address := top + 1;     { address := $$top }
        top := top + length;
        assume top <= max       { $$top <= $$max }
      end
  end;
```

```
procedure release (index, address : integer);
  begin
    memory[address] := pool[index];
    pool[index] := address
  end;
```

The comments above refer to the values of `$$top` and `$$max` used in the code below, which are one greater than those used by Brinch Hansen.

The actions of `allocate` and `release` are intended to be performed by in-line code on entry to and exit from routines; they are given as separate procedures for documentary purposes.

### 3.5. *Modified algorithm*

In the published form of this algorithm, every routine has its own pool of activation records, which can never be reused by other routines. Brinch Hansen leaves this limitation 'as an exercise for the reader'.

Pools can trivially be shared by routines having activation records of the same size, but this would make little difference. An improvement is to round up the size of each activation record to the next higher power of two, and use this power as the index, so all routines with the same *rounded* size of activation record share the same pool. This should greatly increase the reuse of memory, at the cost of an average wastage of 25%.

The current version also uses intermediate sizes of the form $3 \times 2^n$ (giving 2, 3, 4, 6, 8, 12, etc.), so the ratio between successive sizes is roughly $\sqrt{2}$. This reduces the memory wastage to about 14%, at the cost of less reuse by different routines. It is not clear whether this is really useful, but it is an area where a compiler might be able to optimize the allocator.

## 4. Implementation

The SPARC version of the KRoC translator, `octran`, has been modified so that, as an option, it can handle recursive routines. No changes have been made to the compiler itself.

### 4.1. *Syntax*

The syntactic problem is to make it possible for a routine to be called before its name has come into scope.

The trick used here is to leave the external appearance of a recursive routine unchanged, and to declare a dummy routine inside it having the same name prefixed by '`R.`'*.

* Compare the KRoC convention that C function names start with '`C.`'.

For example:

```
PROC Hanoi (VAL INT n, VAL BYTE x, y, z, CHAN OF BYTE out)
  -- Move n discs from x to y via z

  PROC R.Hanoi (VAL INT n, VAL BYTE x, y, z, CHAN OF BYTE out)
    -- Dummy PROC.  Calls to this are mapped into
    -- recursive calls to Hanoi.
    SKIP
  :

  IF
    n > 0
      VAL INT n IS n - 1:
      SEQ
        R.Hanoi (n, x, z, y, out)
        out.string ("Move from tower ", 0, out)
        out ! x
        out.string (" to tower ", 0, out)
        out ! y
        out.string ("*c*n")
        R.Hanoi (n, z, y, x, out)
    TRUE
      SKIP
:
```

(Note that the redeclaration of `n` can work only if it comes into scope at the end of its declaration.)

Calls to routines like `R.Hanoi` are then translated into recursive calls to `Hanoi`. This is possible because, although `Hanoi` is not in scope to occam, it is in the transputer assembly language.

The dummy routine must have *exactly* the same interface as the real routine. This causes difficulties with 'hidden parameters' – in particular, the pointer to the vector space, and static links used to access non-local variables.

Since this syntactic trickery is entirely local to the routine, external calls are perfectly normal; for example, `Hanoi (10, 'X', 'Y', 'Z', out)`.

It would be possible to give the dummy routine the same name as the real one, without any modification, but that would break existing programs.

### 4.2. *Who does what?*

Modified code for calls and returns could be generated in some combination of the following places:

- In the calling routine, in the instructions leading up to the actual call;
- At the beginning of the routine itself;
- In the instructions at the end of the routine leading up to the return;
- In the calling routine, following the return.

Which of these is chosen for this implementation depends on where the relevant information is available to the code generator.

### 4.3. *Who knows what?*

The compiler, of course, knows everything, but not all of this information is accessible to `octran`, which needs:

- The size of the workspace. In the transputer assembly language, routine headers have the following form:

```
L41:    -- PROC Hanoi, lexlevel: 1, WS: 22, VS: 0
```

  where `WS` and `VS` indicate the workspace and vector space required by the routine.

- The number of parameters passed to the routine. This is available at the point of call, though not at its entry point. If it is not more than three, it is the current depth of the integer evaluation stack; if it is four or more, the additional parameters will have been stored in the workspace, marked with a `parameter` comment:

```
ldl     4       -- CHAN out
stl     1       -- parameter
ldc     90
stl     0       -- parameter
```

  Counting these comments gives the total number of parameters.

- The name and label of the routine. Calls refer to the assembly-language label (`L41` in this case), not the occam name (`Hanoi`), although this is included as a comment:

```
call    L41     -- Call Hanoi
```

  The mapping between names and labels can be found from the header, or from the `external` directive for a separately compiled routine.

For separately compiled routines, the size of the workspace is not known at the point of call, so in general the activation record cannot be allocated there.

Inside the routine, the number of parameters is not known, so allocation cannot be done in the entry code either.

The only place where all the necessary information is available is at the point of a *recursive* call, inside the routine.

Hence calls to normal* routines, and normal calls to recursive routines, are unchanged, and pre-existing programs are unaffected. Calling a recursive routine looks (and indeed is) normal to the caller; any syntactic trickery and modified code are internal to the routine. This retains the security of current occam programs. It is also more efficient than using the recursive mechanism for all calls, though this may not be significant.

## 5. Usage

Recursion is enabled with a command-line flag to `octran`; for example

```
octran -X5 hanoi.kt8 hanoi.s
```

where the number after the `X` flag specifies the size of the heap, in megabytes, to be used for allocating activation records.

* Here ‘normal’ means non-recursive, implemented in the unmodified way with statically allocated activation records.

## 6. Code generated

On the SPARC, a normal routine call, such as `Hanoi (10,'X','Y','Z',out)`, is translated as follows. First, any parameters after the third are placed in the workspace:

```
ld      [%l3+16],%l0     !- CHAN out
st      %l0,[%l3+4]      !- parameter
mov     90,%l0
st      %l0,[%l3]        !- parameter
```

and the first three are loaded into the registers of the evaluation stack:

```
mov     89,%l0
mov     88,%l1
mov     10,%l2
```

The transputer `call` is translated into a short sequence of instructions. The workspace pointer is advanced by four words, and the active registers of the evaluation stack and the return address are stored in the space so allocated. The routine is then called, with the return address being saved in the delay slot of the `call` instruction:

```
dec     16,%l3           !- Call Hanoi
st      %l2,[%l3+4]
st      %l1,[%l3+8]
st      %l0,[%l3+12]
call    L41              !+ Hanoi
st      %o7,[%l3]        !*
```

The workspace pointer is restored in the return from the routine:

```
ld      [%l3],%o7
retl
inc     16,%l3           !*
```

A recursive call such as `R.Hanoi (n, x, z, y)` is implemented by a longer sequence. As before, the parameters are set up:

```
ld      [%l3+36],%l0     !- CHAN out
st      %l0,[%l3+4]      !- parameter
ld      [%l3+28],%l0     !- BYTE y
st      %l0,[%l3]        !- parameter
ld      [%l3+32],%l0     !- BYTE z
ld      [%l3+24],%l1     !- BYTE x
ld      [%l3+8],%l2      !- INT n
```

The pool for this size of activation record is checked (instructions in the delay slots of branches make this code harder to follow):

```
ld      [%l7+$$pool+36],%i0
tst     %i0              !- Call R.Hanoi
bne,a   1f
ld      [%i0],%i2        !@
```

If the pool is not empty, it is used; otherwise a new record is claimed from the free space (if this is possible):

```
        ldd     [%l7+$$top],%i0 !+ and $$max
        add     %i0,128,%i2     !+ size of record
        cmp     %i2,%i1
        tgu     23              !+ MEMORY
        ba      2f
        st      %i2,[%l7+$$top] !*

1:      st      %i2,[%l7+$$pool+36]
2:      st      %l3,[%i0]       !+ save Wptr
```

Additional parameters after the third must be copied from the old workspace to the new (with the workspaces suitably aligned, this can be done two words at a time):

```
        ldd     [%l3],%i2       !+ parameters
        std     %i2,[%i0+112]
```

The workspace pointer is set to point to the new activation record:

```
        add     %i0,96,%l3      !+ new Wptr
```

Saving the stack registers and the call are as normal:

```
        st      %l2,[%l3+4]     !+ Areg
        st      %l1,[%l3+8]     !+ Breg
        st      %l0,[%l3+12]    !+ Creg
        call    L41             !+ Hanoi
        st      %o7,[%l3]       !* save Iptr
```

After the return, the activation record is released to the appropriate pool:

```
        sub     %l3,112,%i0     !+ release
        ld      [%l7+$$pool+36],%i1
        st      %i0,[%l7+$$pool+36]
        ld      [%i0],%l3       !+ restore Wptr
        st      %i1,[%i0]
```

Since the return instruction has already adjusted the workspace pointer by the normal amount, the adjustment here is by 16 bytes (4 words) more than in the call.

## 7. Limitations

This is not a fully general implementation of recursion in occam. Indeed, this would probably not be possible without some help from the compiler itself.

### 7.1. *Vector space*

At present, the vector space used by a routine is not included in the activation record. There is a further complication that the pointer to the vector space is passed in one way to externally visible routines, and in a different way to all others. This problem could probably be solved, but it would be tedious. At present, variables that the compiler would normal place in vector space can be explicitly `PLACE`d `IN WORKSPACE`.

### 7.2. *Non-local variables*

To ensure that the static links are passed in the recursive call, there would have to be dummy references to any non-locals in the dummy routine.

### 7.3. *Function calls in actual parameters*

If any of the actual parameters itself involves a function call, it is impossible to distinguish which routine the `parameter` comments refer to. This is only a problem if either the function, or the routine of which it is a parameter, has more than three parameters – so `R.Ackermann (m - 1, R.Ackermann (m, n - 1))` works.

### 7.4. *Mutual recursion*

The present implementation requires that the entry point of the recursive routine must be known at the point of the recursive call. This is impossible with mutual recursion, though a more complicated treatment could solve this problem.

### 7.5. *Memory usage*

The activation record used for a recursive routine includes space not only for the routine itself, but also for all the routines that it might call, even when it does not.

### 7.6. *Multiprocessors*

None of these problems is difficult in principle. However, Brinch Hansen's other 'exercise for the reader', implementing the algorithm for shared-memory multiprocessors, is much harder. As he says, it involves claiming and releasing locks on entry to and return from routines, which is inevitably expensive.

## 8. Results

A normal routine call on the SPARC costs three instructions, plus one for each parameter up to three. (Further parameters are stored in the workspace before the call.) A return also takes three instructions. The code above, including releasing the activation record after the return, is 17 instructions longer than a normal call. Further parameters after the third cost about one instruction each.

Eleven extra instructions are executed if reusing a record (which should be the common case); 15 if creating a new one (including a check for running out of memory). This is rather more than the 'three or four' suggested by Brinch Hansen, but still small compared with the body of most routines.

Measurements indicate that, provided the depth of recursion is reasonable, the overhead of a recursive call is less than 50% greater than that for a normal call (for example, 61 ns compared with 42 ns). For routines involving any significant amount of computation, this is negligible. Very deep recursion (or very large activation records) take the allocation out of the cache, and the overheads rise significantly, but this would also happen in analogous situations without recursion.

The smallest possible activation record uses 12 words (for 64-bit alignment), or 48 bytes. The present implementation is restricted (by the 13-bit immediate-value field of SPARC instructions) to activation records of less than 4096 bytes. This could easily be extended, but it is adequate for the largest workspace in the `cgtest` suite. Hence only a small number of pools (less than twenty) are required.

Using a fixed set of sizes means that the pointers to the various pools can be set up statically, without any help from the compiler; hence there are no problems with separate compilation.

This implementation is not optimal. Ideally, with more information from the compiler, parameters after the third should be stored directly in the new activation record, and one instruction in the return sequence could be eliminated, but these points make very little difference.

### 8.1. *QuickerSort*

For a realistic application, the occam implementation of the *QuickerSort* algorithm [5] described in [6] was modified to use the system described here.

Parallel recursion to depth $d$ with this algorithm requires $2^d$ processes, which becomes excessive even for reasonable depths. For this reason, an additional parameter is used to keep track of the depth of recursion, and the routine switches from parallel to sequential recursion when this reaches some suitable limit:

```
PROC quicker.sort ([]INT X, VAL INT quicker.threshold)

  PROC q.sort ([]INT X, VAL INT bottom, top, threshold, depth)
    #PRAGMA SHARED X

    PROC R.q.sort ([]INT X, VAL INT bottom, top, threshold, depth)
      SKIP              -- Dummy for recursion
    :

    VAL INT n IS top - bottom:
    IF
      n < threshold
        ...  insert sort
      TRUE
        INT lo, hi:
        SEQ
          ...  partition X
          IF
            depth > 0
              VAL INT depth IS depth - 1:
              PAR
                R.q.sort (X, bottom, lo - 1, threshold, depth)
                R.q.sort (X, hi, top, threshold, depth)
            TRUE
              SEQ
                R.q.sort (X, bottom, lo - 1, threshold, 0)
                R.q.sort (X, hi, top, threshold, 0)
  :

  q.sort (X, 0, (SIZE X) - 1, quicker.threshold, 10)
:
```

This code was run as shown here, and also without the parallelism (effectively with a depth parameter of zero). It was compared with a routine equivalent to that described in [6], running on a single processor. The speeds were all the same to within the accuracy of measurement, but the recursive code was smaller by a factor of more than 20, because there was only a single instance of the routine, rather than the many required when using the mechanism described in § 2.2.

### 8.2. *Sieve of Eratosthenes*

A theoretical version of the sieve of Eratosthenes for generating prime numbers is given in recursive occam in [7]. It now runs as a real program:

```
PROC from (VAL INT start, step, CHAN OF INT c)

  INT count:
  SEQ
    count := start
    WHILE TRUE
      SEQ
        c ! count
        count := count + step
:

PROC filter (VAL INT n, CHAN OF INT in, out)

  WHILE TRUE
    INT a:
    SEQ
      in ? a
      IF
        (a\n) <> 0
          out ! a
        TRUE
          SKIP
:

PROC sieve (VAL INT count, CHAN OF INT in, out)

  PROC R.sieve (VAL INT count, CHAN OF INT in, out)
    SKIP                -- Dummy for recursion
  :

  IF
    count = 0
      id (in, out)      -- Just copy in to out
    TRUE
      INT n:
      SEQ
        in ? n
        out ! n
        CHAN OF INT c:
        PAR
          filter (n, in, c)
          R.sieve (count - 1, c, out)
:
```

```
PROC primes (CHAN OF INT out)

  -- Generates primes up to one million
  -- (Square of 169th prime > 1,000,000)

  VAL INT LIMIT IS 169:
  SEQ
    out ! 2
    CHAN OF INT c:
    PAR
      from (3, 2, c)
      sieve (LIMIT, c, out)
:
```

## 9. Conclusions

This experiment has shown that recursion can be implemented efficiently in a concurrent language like occam, using Brinch Hansen's algorithm for allocating memory.

Brinch Hansen suggests that this mechanism should be used for all routine calls, so that, as with most sequential languages, no special action would be required for recursion. Although it is imposed by accidental details of the implementation, the method used here, in which only recursive calls of recursive routines are treated specially, has a small advantage in efficiency and retains the security of occam for non-recursive routines.

A proper implementation would require some change to the syntax of the language, and changes in the compiler to deal with some difficult points, but this does not affect these conclusions.

## 10. References

[1] David C. Wood and Peter H. Welch. The Kent Retargetable occam Compiler. *Proceedings of WoTUG-19: Parallel Processing Developments*, edited by Brian C. O'Neill. IOS Press, 1996. ISBN 90–5199–261–0.

[2] Per Brinch Hansen, Efficient Parallel Recursion, ACM SIGPLAN Notices, Volume 30, No 12, December 1995.

[3] Michael Poole, Fixed Maximum Depth Recursion in occam. *WoTUG Newsletter* No 16. January 1992.

[4] Andy Whitlow, Occam Run-time Model Specification, INMOS Limited, June 1990.

[5] C. A. R. Hoare. Quicksort. *Computer Journal* (5):10–15, 1962.

[6] Kevin Vella and Peter H. Welch. CSP/occam on Shared Memory Multiprocessor Workstations. *Proceedings of WoTUG-22: Architectures, Languages and Techniques*, edited by Barry M. Cook. IOS Press, 1999. ISBN 90–5199–480–X

[7] James Moores. CCSP – A portable CSP-based run-time system supporting C and occam. CSP/occam on Shared Memory Multiprocessor Workstations. *Proceedings of WoTUG-22: Architectures, Languages and Techniques*, edited by Barry M. Cook. IOS Press, 1999. ISBN 90–5199–480–X