KROC — The Kent Retargeted Occam Compiler

David C. Wood
Computing Laboratory
University of Kent at Canterbury

ABSTRACT

This paper describes an implementation of *occam** for the SPARC architecture. It uses a standard compiler to generate Transputer assembly language, which it translates into assembly language for the SPARC. This is then assembled and linked with a kernel that performs the necessary scheduling between different processes. This process provides an easy method of porting occam to other machines.

NOTE: This is a **DRAFT** paper and may not represent the current state of the KROC system and thus may be incomplete or wrong.

1. Introduction

Historically, the occam language has been closely associated with the Transputer. The aim of the *Occam For All* project is to make it available on a wider range of machines. As part of this project, KROC is intended to provide a simple way of compiling and running occam programs on a number of target machines. A version for the SPARC processor under UNIX† is now working, and others for the DEC Alpha, Motorola 68000, and Motorola 68HC11 microcontroller are under development. Other proposed targets are the PowerPC and the Analog Devices 21060 SHARC digital signal processor.

2. Strategy

KROC uses the INMOS‡ occam compiler to produce Transputer assembly language, which it translates into that for its target machine, currently the SPARC. This approach means that any suitable occam compiler can be used, without modification; for example, no changes were required to run occam 2.1.

The translator maps the Transputer registers (the evaluation stacks, and the workspace, process queue, and timer queue pointers) onto SPARC registers in a way that will require little modification for other processors with a general-register architecture. It makes no use of the SPARC register-window mechanism, except that it does not need to save its registers when calling external functions.

The translated code is assembled and linked with a small kernel. This is written in the assembly language of the target machine, to give direct access to the registers, and also for efficiency.

Communications with UNIX, for input and output, timing, etc., are provided by additional functions written in C.

The mechanics of this process are hidden in a shell file, *kroc*. This takes an occam source file (for example, *prog*.occ), compiles it to Transputer assembly language (*prog*.t) using *oc* with the *-zb* option, translates it into SPARC assembly language (*prog*.s), assembles and links it using *gcc* (*prog*), and finally runs it. (These intermediate files were very useful during the development of KROC, and may be used for debugging. They can now be deleted automatically.)

---

* Occam is a trademark of the SGS-THOMSON Microelectronics Group.
† UNIX is a trademark of AT&T Bell Laboratories in the USA and other countries.
‡ INMOS is a member of the SGS-THOMSON Microelectronics Group.

If any step of this process fails, the shell program aborts.

3. The Kernel

This is mostly concerned with scheduling occam processes, and most operations that involve manipulation of the process queues are implemented as calls to the scheduler. (Since `startp` and `runp`§ are very short, they are translated in line.)

In particular, it provides communications on occam channels, both between processes and with the external world, and timeouts. The algorithms for these operations were designed in occam (with extensions to handle pointers) and then coded in assembly language, with the occam included as comments. This will simplify rewriting for other machines.

The code used for communicating between processes is also used for block move instructions (`move` and `move2dall`). It is optimized to the extent that it identifies blocks consisting of a whole number of words, aligned on word boundaries, which can easily be copied very fast. Other cases are complicated by the endian problem.

The kernel also includes code for allocating workspace, initialization of registers, setting up the channel parameters, and entering the occam program at the label *`$$entry`*, which is inserted by the translator.

On completion, or if it detects deadlock, the scheduler returns the `ErrorFlag` as its result.

The assembly-language kernel is about 1000 lines long (nearly half of them comments), with another 700 lines of C. These produce less than 1800 and 7000 bytes of code, respectively.

4. External Channels

Three external channels, corresponding to *`stdin`*, *`stdout`*, and *`stderr`* in UNIX, are provided. Programs must start with a heading of the form

```
PROC program.name (CHAN OF BYTE in, out, error)
```

If an occam process were to input from the keyboard by calling a simple UNIX input function, the whole program would block, which is unacceptable in occam. For this reason, KROC uses two UNIX processes: the *`parent`*, which calls the input routine, blocking until a character arrives; and the *`child`*, which contains the occam program.

When a key is typed, the parent process sends a signal to the child, and the character is communicated through a pipe. A signal handler in the child sets a flag, which is tested in the scheduler. When the character is received by the occam program, an acknowledgement is sent, and another character can be read. The overhead in the scheduler is thus extremely low, compared with superficially simpler mechanisms that would require very expensive UNIX system calls.

Timeouts, using the UNIX alarm-clock mechanism, are handled similarly.

So that external communications behave exactly like those between processes, the kernel contains small pseudo-processes to handle input from the keyboard and output to the screen.

5. The Translator

This takes readable Transputer assembly language and translates it into SPARC assembly language.

With two exceptions (conditionals and floating-point rounding), it works on one instruction at a time.

It expects its input to be in the form generated by the compiler from occam; so `pfix`, `nfix`, `opr`, and `fpenter` instructions do not appear explicitly, but are implied by the use of large operands and secondary instructions. This makes translation somewhat simpler.

It also assumes that the code satisfies certain conditions; in particular, that each `PROC` and `FUNCTION` has a single return, at the end. Hence, it may not be able to handle code produced from `ASM`

---

§ Occam and Transputer names are here printed like `Transputer`; UNIX and SPARC names like *`SPARC`*.

(or GUY) statements.

In a few situations, the translator has to read the comments provided by the compiler; in particular:

- To identify PROCs and FUNCTIONs;
- To identify the entry point of the program (the PROC at lexical level zero);
- To find out how much workspace and vector space are required;
- To identify stubs for external routines;
- To keep track of the file name and line number of the occam source, for reporting errors.

5.1. *Architectures*

Both the Transputer and the SPARC are 32-bit machines, but their architectures are very different. The Transputer is rather unusual —

- Computation is done with two evaluation stacks, one for integers and one for floating-point numbers, each three registers deep. Most instructions manipulating data have no explicit operands.
- For scheduling many processes and communicating between them, it has a microcoded kernel, and special registers to maintain queues of processes.
- Local memory is accessed through a workspace pointer.
- Addressing is little-endian.

The SPARC is a fairly conventional RISC machine —

- It has a large set of general registers, organized in overlapping windows, so that 32 of them are accessible at any time. These are divided into four sets of eight, global, input, local, and output, so that the output registers of one window are the input registers of the next. There are also 32 global floating-point registers.
- Most instructions operate on three registers, or two registers and a literal.
- The instruction immediately following a transfer of control, called the ‘delay slot’, normally executes, but in conditional branches it can be ‘annulled’ if the branch is *not* taken.
- Addressing is big-endian.

The different endianism is a problem, discussed below.

5.2. *Translation*

Many Transputer instructions translate into single SPARC instructions. This may be a misleading comparison, as primary Transputer instructions with small operands (zero to 15), and the commonest secondary instructions, occupy single bytes, while all SPARC instructions use 32-bit words. However, primary instructions with larger operands may need several prefixes, most secondary instructions need two bytes, and some floating-point instructions need three or even four.

SPARC instructions can include constant operands of up to 13 bits; larger values are handled by the ‘synthetic instruction’ *set*, which normally corresponds to two real instructions. (These constant operands are signed, which in this context very often wastes one bit, since many instructions come in pairs, like *add* and *sub*, *and* and *andn*, and so on.) For example,

```
ldl     1000
ldl     10000
```

translates to

```
ld      [%l3+4000],%l0
set     40000,%i0        !+ big operand
ld      [%l3+%i0],%l1
```

(Comments in the Transputer source are passed on into the SPARC assembly language, with the initial -- replaced by !-. The translator also adds comments of its own. These usually start with !+, but delay and annul slots are marked with !* and !@, respectively.)

Some operations are complicated by trivial differences in definition between corresponding instructions on the two machines. For example, occam allows a shift of 32 bits on an `INT32` variable, giving a result of zero, but shift counts on the SPARC are taken modulo 32, so an extra test has to be included, giving four instructions instead of one.

A very rough estimate is that the number of instructions in the SPARC translation is about 70% greater than in the original Transputer code; and, since SPARC instructions are four bytes long, while most of those on the Transputer are only one or two, the code expands by a factor of about five. This emphasizes the fact that Transputer code is remarkably compact, and that RISC architectures like tend to have a low code density.

A comparison of numbers of cycles is more favourable. Many Transputer instructions take several cycles, while most on the SPARC execute in only one. For example, `call` takes seven cycles, while its translation takes from three to six. Loading and storing bytes, `lb` and `sb`, take five cycles, while, even with the overhead of reversing the endianism, the translation takes only two.

A few instructions translate into quite long sequences. Probably the most important of these are `endp` (seven instructions) and `lend` (ten). The longest is `disc` (17).

The translator makes some unnecessary 'optimizations'. For example,

```
ldc     0
```

is translated as

```
clr     %l0
```

although this is exactly equivalent to

```
mov     0,%l0
```

This is partly to make the translated code more readable, but will also be useful when translators are written for other target machines.

5.3. *Registers*

The registers of the Transputer integer evaluation stack (`A`, `B`, and `C`) are mapped statically onto three SPARC registers (*%l0*, *%l1*, and *%l2*) by the translator, rather than dynamically as the program runs; so, for example, the Transputer code

```
ldc     0               -- A = 0
ldc     1               -- A = 1, B = 0
ldc     2               -- A = 2, B = 1, C = 0
```

becomes

```
clr     %l0             !- A = 0
mov     1,%l1           !- A = 1, B = 0
mov     2,%l2           !- A = 2, B = 1, C = 0
```

on the SPARC. The floating-point stack is handled similarly.

Because the translator knows the state of the evaluation stacks, it can detect underflow or overflow, and give an error message. Also, it can generate code to store only the active registers on a `call` instruction.

Since `FUNCTION`s may return their results on the stacks, the translator records the state of the stacks at the return from a `FUNCTION`, and restores it after the `call`.

The Transputer workspace pointer (`Wptr`) is also represented by a SPARC register, *%l3*.

The instruction pointer (`Iptr`) corresponds, of course, to the the program counter (*PC*). There is a small complication here, because `Iptr` always points to the next instruction, while *PC* points to that currently executing. When values of `Iptr` are saved in the workspace, a consistent convention is needed. For the `ldpi` (load pointer to instruction) it must be 'adjusted' to point to the next instruction, because it is used in generating the addresses of data as well as code. It follows that calls to the scheduler (such as `in` and `out`) have to follow the same convention, and this is done by incrementing the link register by eight (two instructions) in the delay slot of the *call*; for example:

```
        call    $$in
        inc     8,%o7              !* adjust
```

This differs from the normal SPARC convention, in which the adjustment is made in the return instruction.

However, for translating the Transputer subroutine instructions (`call` and `ret`), the ‘unadjusted’ *PC* is used for simplicity.

The process queue pointers, `Fptr` and `Bptr`, are kept in the SPARC registers *%l4* and *%l5*, and the timer queue pointer, `Tptr`, in *%l6*.

To simplify the scheduler, *%l7* is used as a pointer to an area of frequently accessed variables.

The Transputer makes extensive use of the number `MostNeg`, or `MinInt`, (hexadecimal 80000000) so it is convenient to hold this in a register, *%i4*.

The `ErrorFlag` and `HaltOnErrorFlag` are represented as bits in *%i5*.

Four registers, *%i0* to *%i3*, are available for use as temporary variables, both by the translated code and by the kernel.

5.4. *Conditionals*

The most obvious inefficiency of translating instructions one at a time arises in conditionals. On the Transputer, a comparison generates a Boolean value, represented by zero or one, on the evaluation stack, and this is tested by the conditional jump instruction, `cj`.

For example,

```
        gt
        cj      L666
```

would produce something of the form

```
        cmp     %l1,%l2
        bg,a    1f
        mov     1,%l1              !@ true
        mov     0,%l1              !+ false
1:
        tst     %l1
        be      L666
        nop
```

with an unnecessary Boolean value being generated.

To avoid this, the translator defers generating the Boolean until the next instruction has been identified, and omits it if it finds a `cj`, producing the appropriate conditional branch instead:

```
        cmp     %l1,%l2            !+ remember GT
        ble,a   L666               !+ was GT
        mov     0,%l1              !@ make sure it's false
```

The instruction in the delay slot of the branch is sometimes needed in evaluating Boolean expressions; it involves no extra cost, since otherwise a *nop* would be required.

There is a further special case in the use of `eqc 0` for inverting conditions, which generates no extra code:

```
        gt
        eqc     0
        cj      L666
```

gives

```
        cmp      %l1,%l2         !+ remember GT
!       invert condition
        bg,a     L666            !+ was LE
        mov      0,%l1           !@ make sure it's false
```

5.5. *Memory*

The whole of the occam workspace is represented by a single SPARC stack frame, which is organized as follows, from low address to high:

- A few words, accessed relative to the SPARC frame pointer, `%fp`, needed for some floating-point instructions, because the only path between the floating-point and integer processors on the SPARC is through memory;
- The workspace of the main occam process, including a pointer to the vector workspace and the file parameters corresponding to `stdin`, `stdout`, and `stderr`, set up by the kernel;
- The rest of the occam workspace, arranged exactly as on the Transputer;
- The vector space, if any;
- Ninety-two bytes required by the SPARC for dumping registers etc.

The translator generates a symbol, `$$space`, the value of which is used by the kernel to allocate this space.

Variables shared between the assembly-language kernel and the C routines, together with other workspace used by the scheduler, are in the UNIX `.data` area. For speed of access, a SPARC register (`%l7`) is used as a pointer to this area.

5.6. *Error Detection*

The code needed after checked arithmetic operations to implement the full `ErrorFlag`–`HaltOnErrorFlag` mechanism is considerable, so four alternative levels of checking are provided:

- None;
- Setting `ErrorFlag`, but no `HaltOnErrorFlag`;
- Trap on error (this is the default);
- `ErrorFlag` and `HaltOnErrorFlag`.

A trap handler in the child process reports fatal errors.

5.7. *Translation Errors*

The translator checks for a number of errors; for example, underflow and (unexpected) overflow of the evaluation stacks, unrecognized and unimplemented instructions, calls to undeclared `PROC`s and `FUNCTION`s, and errors in the sizes of floating-point operands.

In each case, it prints the offending line, with its line number in the Transputer assembly-language file, and the line number and name of the occam source file, and then exits.

5.8. *Speed*

Of the steps involved in generating an executable file, translation is usually about twice as fast as compilation, and somewhat faster than assembly, so it contributes less than a quarter of the total.

6. Separate Compilation

There are three situations where occam programs may call separately compiled routines.

In order to call such routines, the compiler needs certain information about them; in particular, the number and type of their parameters and results, and the amount of workspace that they (and their descendants) use. This is difficult for KROC.

6.1. *Compiler Functions*

The occam compiler sometimes generates calls to 'compiler functions'; for example, for `INT16` and `INT64` arithmetic. In the Transputer assembly language, these functions appear as stubs, commented with their names, which would be filled in by the Transputer linker. The translator replaces calls to these stubs by calls directly to the functions.

The names of these compiler functions always contain a `%` character. Since this is not acceptable to the SPARC assembler, it is replaced by a *$*. The rest of the name, following this character, seems never to be significant, and can be truncated for simplicity.

The bodies of these compiler functions are written in occam. They can be translated and assembled in the usual way, and included in the library of run-time support routines used by KROC.

Some of these functions contain `ASM` sections. In some cases these violate the assumptions made by the translator, so it is necessary to rewrite them directly in SPARC assembly language.

Since there is a limited number of them, the necessary information about these functions can be built into the translator.

6.2. *Separately Compiled Occam*

Libraries of occam `PROC`s and `FUNCTION`s can be included in a program in two ways; either in source form, using the `#INCLUDE` directive, or pre-compiled, with `#USE`.

The first is invisible to KROC, but the second involves the use of 'TCOFF' files for the pre-compiled libraries, which contain the information the compiler needs about the parameters and results.

This has some problems for KROC. The TCOFF files also contain Transputer binaries, which are a waste of space; and the compiler will not produce both TCOFF and assembly-language output at the same time, so libraries have to be compiled twice.

An alternative scheme is possible, based on the method used for calling C functions from occam. It has the advantage that any external function, whether written in occam or in C (or SPARC assembly language, or Fortran, . . .), is accessed in the same way.

- The library is compiled, translated to SPARC assembly language, and assembled;
- Occam 'prototypes' are created, using the `PROC` headers from the occam source, and the workspace information provided in the assembly-language files;
- The library can then be compiled, using the `#INCLUDE` directive to access the 'prototypes', and linked to the library.

In either case, there is the problem that the translator does not know about the results of `FUNCTION`s.

6.3. *Calling C Functions*

KROC provides a mechanism for calling functions written in C.

For a function with the C prototype

```
int some_thing (int this, int that);
```

an occam 'prototype' is required, of the form

```
#PRAGMA EXTERNAL "PROC C.some.thing (INT result, VAL INT this, that) = 0"
```

Since a C function are likely to have side effects, it is represented as a `PROC` with an extra `result` parameter.

The initial `C.` is a naming convention to allow the translator to generate appropriate calling code. Underscore characters in C names correspond to dots in occam names, since neither character is allowed in the other language.

The zero at the end is the number of words of occam workspace required by the function. Since C functions create new SPARC stack frames for their workspace, none is needed.

The two languages have different mechanisms for passing parameters. To convert between them, an interface function is required. This can be written in C, so the details of the C mechanism are handled by the compiler. This function is passed a single parameter, which is the region of the occam workspace containing the occam parameters, treated as an array of *word*s, where the type *word* is a *union* of all the data types that can be represented in a single word, here integers (*i*) and pointers to integers (*ip*):

```
void _some_thing (w)
word w[3];
{       *w[0].ip = some_thing (w[1].i, w[2].i);
}
```

An interface to part of the *wn* library, which provides access to some of the facilities of *X11*, has been written to demonstrate this mechanism.

Because of the endian problem, parameters of some types have to be converted for one format to the other. Simple floating-point variables can be converted in the interface routines, but other types, such as character strings and arrays of REAL64s, may be processed by several C functions, so it is more efficient to convert them only when required. Special functions are provided for this.

Routines written in other languages, such as Pascal or Fortran, can be accessed through C interface functions. Hence, for example, libraries like NAG could be used.

Assembly-language routines do not need separate interface functions; they can be written using either the normal occam conventions or those of the C interfaces.

7. Problems and Solutions

7.1. *Endianism*

The Transputer is a little-endian machine, and the SPARC is big-endian.

In the Transputer, arrays of BYTEs and BOOLs are stored in single bytes, but simple variables of these types are held in words. For parameters passed by reference, the object has to be in the right (little) end of the word. Hence all accesses to bytes have to be modified (by inverting the low-order three bits of their addresses). This is quite expensive in block moves (move instructions and in−out communications).

INT16s are treated similarly.

All constants, of whatever size, are compiled into .BYTE directives, in little-endian order. Since there is no way of knowing how they are to be used, they have to be translated consistently.

Hence groups of four .BYTEs are translated into *.word* directives, in reverse order.

It would be more efficient if REAL64s (and INT64s) could be accessed using the SPARC *ldd* and *std* instructions, but they are in the wrong order. However, this is impossible anyway, because of the different alignment requirements.

7.2. *Floating Point*

Since both the Transputer and the SPARC conform to the IEEE 754 floating-point standard (and neither supports extended precisions), numerical results should be identical. Preliminary tests confirm this.

The Transputer floating-point evaluation stack is handled in the same way as the integer stack, being mapped statically onto six SPARC floating-point registers, two for each 64-bit item.

The precision of the Transputer floating-point registers is dynamic, being recorded in an internal flag associated with each register. This is simulated statically by the translator, which also checks that the precisions of operands match.

Because the SPARC requires 64-bit operands to be aligned on eight-byte boundaries, while the Transputer workspace is aligned only to four bytes, the two halves of REAL64 variables have to be loaded and stored separately. This also avoids the problem of incompatible endianism.

A number of constants used by Transputer floating-point instructions, such as 0.0 and 0.5, are held in SPARC floating-point registers.

The Transputer rounding mode defaults to `ToNearest` after every floating-point instruction. Since modifying the SPARC floating-point state register, *%fsr*, which controls the rounding mode, is rather complicated (it may have to be stored in memory, loaded into an integer register for manipulation, returned to memory, and re-loaded), the translator keeps a record of the current rounding mode, and changes it only when necessary.

By default, the floating-point errors *invalid operand*, *overflow*, and *division by zero* (*NV*, *OF*, and *DZ*) cause a trap, but alternatively the Transputer `FP_Error_Flag` can be simulated by the the accrued exception field, *aexc*, of the SPARC *%fsr*.

The Transputer computes square roots with a sequence of instructions (`fpusqrtfirst`, `fpusqrtstep`, and `fpusqrtlast`). Since the SPARC requires only a single *fsqrts* or *fsqrtd*, only `fpusqrtlast` is translated.

It is not clear from the available documentation exactly what the instructions of the floating-point remainder sequence (`fpremfirst` and `fprestep`) do. At present, only the first is implemented, using a call to the SPARC mathematical library function *_remainder*.

The SPARC has no instruction for moving a double-precision floating-point quantity between registers; the two words have to be moved separately, using *fmovs*. Since KROC keeps the value zero in floating-point registers for other reasons, it seems in the RISC tradition to use something of the form

```
faddd    Source,Zero,Destination
```

Experiments on a range of different SPARC processors confirm that this is up to twice as fast as the conventional method.

The Transputer has an instruction for multiplying a floating-point number by two, `fpmulby2`. On the SPARC, adding a register to itself is as fast or faster, and does not require the value two to be kept in registers.

7.3. *Stack Shuffling*

The implementation of the evaluation stacks in KROC requires that the translator should always be able to deduce how many items are on each stack.

This is usually true, but in some situations redundant information may be left on the integer stack, and the compiler allows this to overflow. The following cases are known:

- Enable and disable in `ALT`s (`enbc`, `enbs`, `enbt`, `disc`, `diss`, and `dist`). The Boolean precondition is not always used.
- Conditional jumps (`cj`). The zero left on the stack when the jump is taken is sometimes used.
- Long arithmetic (`lsum`, `ldiff`, `lmul`, `lshl`, and `lshr`). The second word not always required.
- Range checks (`csub0` and `ccnt1`). The value checked is not always used. These are often commented with '`throw away the result`', but not always.
- Addition is sometimes used for checking (`add` and `adc`). These seem always to be commented '`check for replicator overflow`'.
- Indexing arrays of timers (`ldl`). This is commented '`throw away the result`'.

In each of these, The translator records the fact that one or more items on the stack may not be required. If the compiler provides comments that clarify the situation, the translator takes account of them. Then, if it encounters a situation that would cause the stack to overflow, it 'shuffles' the registers.

It tries to keep track of how many items may be lost in this way, but the process is somewhat unsatisfactory, as it could allow translation errors to go undetected, and also introduces a small inefficiency. For this reason, the number of times that the stack has been 'shuffled' is reported at the end of translation.

Typically, it happens only once in several hundred instructions.

A particular problem arises with labels, in that the stack level may appear to be different depending on whether the label was reached directly or from a jump. In this case, the translator assumes that the lowest level is correct, and introduces a shuffle into the appropriate path.

The same thing happens when there are two or more jumps to the same label.

On very rare occasions, the translator may be unable to resolve a conflict in apparent stack levels; it then prints a message and continues. (This has only happened once, and the program concerned runs correctly.)

There is a similar problem with the floating-point stack (`fprem` sequence). Here one result is probably never used by occam, but may be needed for range-reduction in the mathematical functions, presumably in `ASM`.

8. Future Work

Work is in progress on a number of improvements to the present KROC, as well as the versions for the Alpha and other machines.

8.1. *Multi-Processor Systems*

A major improvement would be to be able to run occam programs distributed over a network of SPARC workstations (or indeed heterogeneous networks, when implementations for other processors are complete).

The system under development is based on the design of the Virtual Channel Router (VCP) of the T9000 Transputer, and will provide communications between machines over an ethernet.

8.2. *Separate Compilation*

The main difficulty that KROC has here is knowing the state of the evaluations stacks on return from an external `FUNCTION`, or equivalently the number and type of the results returned.

An inelegant solution is to assume that the stacks are fully occupied, but to mark them as shufflable.

Although it contradicts one of the basic ideas behind KROC, that of treating the occam compiler itself as a ‘black box’, it is hoped to modify the compiler to provide this information in a comment, which the translator could read.

8.3. *Calling C Functions*

The interface functions described above currently have to be written by hand. A tool is being written that, given a C function prototype, will generate the occam `#PRAGMA EXTERNAL` and the C interface function.

The current mechanism provides a *procedure* interface; it would also be very useful to have a *process* interface, to enable occam to call, for example, C input functions without blocking.

8.4. *File Access*

At present, KROC can only access files through the standard UNIX streams, *`stdin`*, *`stdout`*, and *`stderr`*. An efficient method of reading and writing general files is clearly required.

8.5. *Optimization*

The process by which KROC works — compiling for the Transputer, and then translating the instructions one at a time — cannot be expected to produces efficient code for a machine like the SPARC, which has a very different architecture. It is perhaps surprising that it is as good as it is.

Various approaches to optimization are under consideration. The most promising is to use a peephole optimizer on the SPARC assembly language, since there are a number of common sequences of instructions that could easily be improved.

The UNIX assembler, *`as`*, has an optional optimizer, but it is intended for use only with the output from UNIX compilers, and makes assumptions about the code that KROC cannot satisfy.

Another possibility is for the translator to keep a record of the values of constants in Transputer registers, rather than translating `ldc` instructions literally. In many cases, these values could then be used as immediate-mode operands. Experiments suggest that the efficiency gained by this process would not be great, while the cost in complexity in the translator would be considerable.

The block move routine in the scheduler, used for inter-process communications and by the `move` and `move2dall` instructions, is efficient for copying blocks of words, but, because of the endian problem, is much slower otherwise. It could be made as efficient as copying data in the normal order, using shift instructions in a way similar to that used in the UNIX *memcpy* routine.

8.6. *Debugging*

At present, there are no facilities for debugging.

An option in the translator is being developed that will plant extra instructions in the translated code to record information (for example, the occam file name and line number) for debugging. This could then be used by a post-mortem analyser.

8.7. *Priorities*

The only feature of occam not supported is `PRI PAR`, which is treated as `PAR`.

9. Performance

9.1. *CG Tests*

The ‘CG’ test suite contains 27 test programs for occam 2, with another 18 for occam 2.1, consisting of about 56,000 lines of occam.

KROC passes all of these, with the exception of `cgtest19`, which uses compiler functions with `ASM` sections that the translator cannot handle, and which have not yet been re-written in SPARC assembly language.

9.2. *DAXPY*

This is one of the standard BLAS routines, which adds a multiple of one array to another. In occam:

```
PROC daxpy ([]REAL64 a, VAL []REAL64 b, VAL REAL64 c)
  SEQ i = 0 FOR SIZE a
    a[i] := a[i]+ (b[i]*c)
:
```

Several different versions of this have been run:

(a) In occam, as above;

(b) In occam, with the loop unrolled 16 times;

(c) In occam, with the loop in `ASM`, unrolled 16 times;

(d) In C, called from occam, with the data in SPARC format;

(e) In SPARC assembly language, with the data in Transputer format, unrolled eight times;

(f) In SPARC assembly language, with the data in SPARC format, unrolled eight times.

These can be compared with the same code compiled by SPOC, the Southampton Partable Occam Compiler, and also run on T800 Transputers.

The absolute speeds, in MFLOPS, vary greatly between different models of SPARC, but the relative times are very similar:

Version	(a)	(b)	(c)	(d)	(e)	(f)
SPARC	2.6	4.1	5.1	3.2	6.1	6.8
SPOC	???	???	???	—	—	—
T800	0.45	0.65	0.70	—	—	—

The straightforward occam implementation includes some redundant range-checking, and, because of the Transputer architecture, has to load and store the floating-point variables one word at a time, and make unnecessary memory accesses. Despite this, it compares quite well with C.

9.3. *The 'Numbers' Benchmark*

The 'Numbers' program consists of a network of four processes that generate the natural numbers. It involves very little computation, but requires eight context-switches per cycle.

On a suitable SPARC, the KROC implementation does this in less than six microseconds (5.6 or 4.7 μs, depending whether the outputs in the 'delta' process are done in parallel or sequentially, respectively). The time on a T800 Transputer is 15 μs.

? *SPOC, with and without gcc optimization*

10. Acknowledgements

11. References