The Kent Retargetable occam Compiler

David C. WOOD and Peter H. WELCH
Computing Laboratory, University of Kent at Canterbury, CT2 7NF
{D.C.Wood, P.H.Welch}@ukc.ac.uk

Abstract: A generic approach to targeting occam* to non-transputer architectures is described. The principle is to build a register-level emulation of the major design elements of the transputer, using native registers of the target hardware, and reuse the standard *Toolset* occam compiler with as little alteration as possible. The porting effort thus reduces to an architectural mapping, rather than involvement in the compiler and code-generator. An immediate payoff comes from the reuse of a well developed and sophisticated compiler. An important scientific question, with relevance to efficient and portable parallel computing, is whether the crucial benefits of transputer architecture (such as the very low overheads for the management of processes and events) can be transferred through such emulation. We report some initial results for SPARC-based targets.

1. Introduction

The multi-processing language occam provides a strong formal basis for the secure and efficient development of high-performance parallel applications. Industrial experience over the past ten years (limited to transputer-based hardware) has amply demonstrated its benefits for the engineering of scalable computing applications across a wide spectrum of problem areas. Methodologies and supporting tools for specific areas (e.g. modelling, real-time control, databases, and hardware design) have been developed, and remain the subject of vigorous and high-quality research.

The *occam-for-all* project [1] aims to build a set of architecturally neutral tools, based on occam, that support the design, implementation, optimization, and maintenance of high-performance applications. Systems constructed with these tools will be portable across all current and foreseen (MIMD) parallel-computing platforms, where they will operate with high levels of efficiency. The programme is motivated by fears that the engineering and commercial benefits previously enjoyed through the use of occam on transputer-based platforms will be denied in the future unless the language is 'opened' to the wider parallel computing community. The positive motivation arises from occam being the only parallel-processing language in widespread industrial and academic use whose semantics is sufficiently well defined, simple, and powerful to play this unifying role.

This paper reports on initial developments of KRoC (the Kent Retargetable occam Compiler), which underlies several of the work packages in *occam-for-all*. KRoC is intended to provide a simple way of retargeting occam to run on different architectures. UNIX† versions for the SPARC and DEC Alpha processors have now been released [2]. Others for the PowerPC, the Motorola 68000, the Motorola 68HC11 microcontroller and the Analog Devices 21060 SHARC digital signal processor are under development –

* 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.

including versions targeted for ‘naked’ processors devoid of operating system. Strategic goals for KRoC include transputer-like overheads for concurrency, access to the full (serial) computational performance of the targeted processor, and minimization of the effort needed for retargeting.

occam-for-all is a collaborative project funded by the (UK) EPSRC as part of its Portable Software Tools for Parallel Architectures programme. Formal partners include the Universities of Kent and Keele, British Aerospace and GEC-Marconi Avionics (end users), and SGS-Thomson Microelectronics and Formal Systems Limited (technology suppliers).

2. Virtual Transputers

The thesis underlying KRoC is that the architectural design of the transputer offers significant benefits for the secure and efficient management of concurrent processes and events. Emulating key elements of that design at the lowest possible level in the target architectures *may* enable those benefits to be reproduced – turning those targets into *virtual transputers*.

The difficulty, however, is to achieve those benefits without losing other important properties of the target (such as high integer and floating-point performance) as a result of this emulation. For example, a key problem when targeting RISC architectures is reconciling the very reduced and stack-oriented register set of the transputer with the very extensive and flat register sets of most RISC designs. Nevertheless, the importance that is beginning to be recognized of low-level concurrency (sometimes known as *lightweight threads*) as a design and implementation route for user applications – not just operating systems – makes the attempt to square this circle worthwhile.

3. Tactics

KRoC uses the SGS-Thomson Microelectronics occam *Toolset* compiler to produce transputer assembly language, which it translates into the assembly language for its target machine (currently the SPARC or DEC Alpha). This approach means that the retargeting effort does not involve issues of compilation and that it is independent of the language being retargeted. Indeed, upgrading the SPARC KRoC system from occam 2 to occam 2.1 [3] involved the retargeters in no work.

A cautionary note, however, should be added that the translator does require its transputer assembly-language source code to adhere to certain standards of good behaviour – not all instruction sequences translate! The *Toolset* occam compilers do conform to those standards, but other language compilers may not. This also has implications for in-line transputer `ASM` or `GUY` code within occam programs, which must also stick to the rules.

The translator maps the transputer registers (the evaluation stacks and the workspace, process, 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 when calling external functions (e.g. UNIX system calls or any user-supplied C code).

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.

The current KRoC releases are built on top of UNIX, through which it has to plough to gain access to timers and external input/output. This is achieved by a small set of procedures, written in C, which are called from the kernel.

The mechanics of this process are managed by a driver program, *kroc*, which –

- compiles an occam source file (for example, *prog*.occ), to transputer assembly language (*prog*.t), using a modified version of the occam *Toolset* compiler;
- translates this into SPARC assembly language (*prog*.s), using the KRoC translator;
- assembles and links this with the kernel to produce the executable (*prog*), using standard GNU and UNIX tools.

The intermediate files may be deleted automatically. If any step of this process fails, the driver program aborts.

4. The Kernel

The KRoC software kernel provides those functions from the transputer's microcode that are not provided by the target hardware. These are mainly concerned with process scheduling, timers, communications, and event handling.

For KRoC applications running on top of host operating systems, there was the possibility of exploiting multi-processing facilities already provided by those systems (e.g. *lightweight threads*). However, this approach was ruled out as being far too heavyweight to support the fine-grained parallelism we wanted to allow occam programmers (and which we believe to be necessary for simple application-oriented design and scalable multi-processor performance).

On the other hand, the transputer algorithms and data-structures for multi-processing are elegant, simple, and secure, and are reasonably well documented [4, 5, 6, 7]. Some details, however, are not publicly available (e.g. reactions to timers and external events), and these have had to be reconstructed.

The KRoC kernel, therefore, follows the transputer mechanisms as closely as possible, although we have allowed ourselves the licence of some experimental deviation here and there.

A crucial constraint we have imposed is never to make UNIX systems calls unnecessarily. System calls take an unacceptably long time to execute. If they were made in the normal flow of the kernel logic, they would completely swamp the kernel overheads we are trying to achieve (which are measured in nanoseconds). This means that, in the main process-management code, we cannot exploit UNIX calls to suppress interrupts, access semaphores, or poll devices. We have also to be careful to avoid system calls that may block, making them only when we know they will complete. Asynchronous input/output calls will help in the future, but those currently available are not portable, and the POSIX.4-compliant ones still remain mostly unimplemented.

Interfacing efficiently with UNIX has been the hardest part of building the KRoC kernel. Implementing the transputer algorithms was much easier. We look forward to future versions of KRoC that target 'naked' processors, where there will be no operating system standing between occam and the external environment.

The kernel reports deadlock and any software errors (e.g. arithmetic overflow, range violations, ...) detected at run-time. On completion, the scheduler returns (to UNIX) the transputer ErrorFlag* as its result. The kernel also includes code for allocating

* occam and transputer names are here printed in a fixed-width Courier font (e.g. ErrorFlag). UNIX and SPARC names are italicized (e.g. *stdin*).

workspace, initialization of registers, setting up the external channel parameters, and, finally, entering the user's occam program. The code used for communicating messages between processes is shared with that 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 be copied very fast. Other cases are complicated by the endian problem (see Section 9.1).

Maintaining a transputer register image using registers in the target architecture means that the kernel needs to be implemented in the assembly language of that architecture. For ease of development, maintenance, and subsequent retargeting to other architectures, the necessary data-structures and algorithms were first written in an extended occam 2.1. These algorithms are purely serial and focus on the management of queues. The extensions allow a Pascal syntax for pointers to make the details easier to develop and comprehend. The space to which these pointers are pointing has, of course, been pre-allocated – it is part of the workspace of each process – so that we do not open any semantic problems associated with dynamic storage. We use serial occam because of the precision of its semantics. An example is given in Section 6.

These algorithms are hand-compiled to target assembly language. The occam sources are left in the assembly-language source as high-level documentation. They have proved invaluable when rewriting the kernel for other machines.

The kernel contains about 1,000 lines of assembly-language source text, nearly half of them comments, which assemble to less than 1,800 bytes of (SPARC) object code. There are also about 700 lines of C source code for interfacing with UNIX, most of which would simply disappear for a version supporting a naked embedded processor. This UNIX interface adds a further 7,000 bytes of object code to the kernel.

5. 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 stdin, stdout, stderr)
```

If an occam process were to input from `stdin` (usually the keyboard) by calling a simple UNIX input function (e.g. *getc()*) and no input were available, that process would be (correctly) blocked. Unfortunately, all other processes would also be (incorrectly) blocked, which is definitely not acceptable. Given that portable system calls for asynchronous input/output do not yet exist (although they have been defined under POSIX.4), 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 and which has to continue running so long as there are runnable occam processes.

The parent process blocks waiting for keyboard input. When a key is typed, it signals the child, writes the character down a pipe to the child, and waits for an acknowledgement. A signal handler in the child sets a flag, which is tested in the scheduler. When this flag is detected, a pseudo-occam process is scheduled that inputs the character from the pipe and outputs it to an ordinary occam channel. When this pseudo-occam process is re-scheduled (by the normal kernel algorithms triggered by the real occam process inputting

* Later versions of KRoC will support the *Toolset* `SP` interface as an alternative.

from that channel), it signals an acknowledgement to the parent (which can now loop to wait for another character) and de-schedules itself. The overhead in the kernel for detecting the presence of external input (testing a flag) is kept extremely low, compared with direct mechanisms that require expensive system calls.

The `stdout` and `stderr` channels are similarly managed through pseudo-occam processes. As far as occam processes are concerned, external communications operate over channels in exactly the same way as internal communications. The kernel does not need to detect whether its channels are being used for internal or external messages. In particular, no special arrangements for handling `ALT`s from the keyboard need to be made.

6. Timers

Timeouts operate in the same way as on the transputer. A time-ordered queue of processes awaiting timeouts is maintained. The UNIX alarm-clock mechanism is invoked whenever the first process in that queue is changed – only one alarm-call is outstanding at any time. A signal handler responds to an alarm-call just by setting a flag, which is tested in the scheduler. Again, the polling overhead in the scheduler reduces to testing this flag, rather than making a system call. When this flag is detected, the kernel responds as described below:

```
--{{{  deal with timer interrupt
INT now:
INITIAL BOOL removed IS FALSE:
SEQ
  tim ? now
  WHILE (Tptr <> NotProcess.p) AND (NOT (Tptr^[Time] AFTER now))
    SEQ
      ...  move first process on timer queue to the run queue
      removed := TRUE
  IF
    (Tptr <> NotProcess.p) AND removed
      ualarm (Tptr^[Time] MINUS now, 0)  -- book new alarm call
    TRUE
      SKIP
--}}}
```

where:

```
--{{{  move first process on timer queue to the run queue
INITIAL POINTER temp IS Tptr:
SEQ
  Tptr := Tptr^[TLink]               -- remove from timer queue
  temp^[TLink] := TimeSet.p          -- record that the timeout expired
  temp^[Time] := now                 -- record the current time
  IF
    temp^[State] = Ready.p           -- the ALTing process has already
      SKIP                           -- been put on the run queue
    TRUE
      SEQ
        temp^[State] := Ready.p      -- mark as runnable
        ...  add on to run queue
--}}}
```

The above occam 2.1 pseudo-code is extracted from the documentation supporting the KRoC kernel (see Section 4). This response is not documented in the transputer data-books.

7. 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` or `FUNCTION` has a single return at the end of its code body and that merging flows of control do not have unreasonably different views of transputer stack levels. Hence, the translator may not be able to handle code derived from untidy `ASM` (or `GUY`) statements.

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

- to identify `PROC`s and `FUNCTION`s;
- 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.

7.1. *Architectures*

Both the transputer and the SPARC are 32-bit machines with 'load–store' architectures; that is, the only references to memory are through load and store instructions, all manipulation of data being done in registers. In other respects they are very different.

The transputer [7] is rather unusual:

- Computation is done with two evaluation stacks, one for integers and one for floating-point numbers. Each stack is three registers deep. Most instructions manipulating data have no explicit operands.
- It has a microcoded kernel for scheduling processes and communicating between them. Special registers maintain the necessary queues.
- Local memory is accessed through a workspace pointer.
- Addressing is little-endian.

The SPARC [8] is a fairly conventional RISC machine:

- It has a large flat set of general registers. These are organized in overlapping windows so that 32 of them are accessible at any time. They are divided into four sets of eight: global, input, local, and output – 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 and is discussed in Section 9.1.

7.2. *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 can be 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 the SPARC 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` (*end-process*, seven instructions) and `lend` (*loop-end*, ten). The longest is `disc` (*disable-channel*, seventeen).

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 may also be useful when writing translators for other target machines.

7.3. *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 being pushed and popped 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 detects underflow or overflow, giving an error message if necessary. Also, it generates code to store only the active registers on a `call`.

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.

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

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`.

The output registers, `%o0` etc., are used as parameters for calling external C functions.

The following table summarizes the use of the SPARC registers by KRoC. Together with an understanding of the layout of the transputer workspace, this is enough to enable assembly-language routines to be written.

SPARC–Transputer Register Mappings			
Class	SPARC name	Transputer name	Description
local	%l0–%l2	Areg–Creg	evaluation stack
	%l3	Wptr	workspace pointer
	%l4	Fptr	front process-queue pointer
	%l5	Bptr	back process-queue pointer
	%l6	Tptr	timer-queue pointer
	%l7		pointer to scheduler variables
input (used as local)	%i0–%i3		temporary workspace
	%i4	MostNeg	0x80000000
	%i5	ErrorFlag, HaltOnErrorFlag	status
	%i6 (%fp)		frame pointer
	%i7		return address to calling function
output	%o0–%o5		parameters/results for C functions
	%o6 (%sp)		stack pointer
	%o7		return address for called functions
global	%g0		zero
	%g1–%g3		used for move2d
	%g4–%g7		reserved by SPARC ABI
	%y		multiply/divide register
floating point	%f0–%f5	FAreg–FCreg	floating-point evaluation stack
	%f6, %f7		floating-point temporary
	%f8, %f9		floating-point zero
	%f9–%f21		floating-point constants
	%f22–%f31		spare
	%fsr	FP_Error_Flag, Round_Mode	floating-point status register

7.4. *Conditionals*

The most obvious inefficiency of translating instructions one at a time arises in conditionals. On the transputer, a comparison generates a Boolean value 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
```

7.5. *Memory*

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

- 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.

7.6. *Error Detection*

The code needed after checked arithmetic operations to implement the full transputer `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.

7.7. *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 inconsistencies 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.

7.8. *Speed*

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

8. Separate Compilation

There are three situations where occam programs may call separately compiled routines. These are described in the subsections below.

In order to translate (transputer assembly-language) calls to such routines, we need to know in what state the calls leave the transputer register stacks. For occam `PROC`s, that state is always empty. However, occam `FUNCTION`s may leave integer or floating-point results on these stacks. The standard *Toolset* compiler does not give this information in its assembly-language output – we are not even told whether the calls are to `PROC`s or `FUNCTION`s.

We are grateful, therefore, to SGS-Thomson Microelectronics for providing the *occam-for-all* project with compiler sources. Another member of the project team (M. D. Poole) has modified these sources to provide the missing information. (The compiler has further been modified so as only to produce assembly-language output plus the header information in 'TCOFF' files necessary for the compiler itself to manage separate compilation. KRoC systems do not, therefore, generate transputer executables. We have permission to release binaries derived from the *Toolset* compiler only to support occam targeted to non-transputer platforms.)

8.1. *Compiler Functions*

The occam compiler sometimes generates calls to 'compiler functions'; for example, for `INT16` and `INT64` arithmetic. In the transputer assembly-language output, these functions appear as stubs, which would be filled in by the *Toolset* linker. The translator replaces calls to these stubs by direct calls to the functions.

The bodies of the 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 them contain `ASM` sections, and in a few cases these violate the assumptions made by the translator, so modified versions have had to be written. For efficiency, some are being rewritten in SPARC assembly language.

8.2. *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 requires the use of the modified occam compiler described above.

8.3. *Calling C Functions*

KRoC provides a mechanism for calling functions written in C. For a function with the C prototype:

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

an occam 'prototype' is required of the form:

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

Since a C function is likely to have side effects (we can never assume it really is a *function* in the mathematical or occam sense), it is represented as a `PROC` with an extra `result` parameter, rather than as a `FUNCTION`. The initial '`C.`' is a naming convention to allow the translator to generate appropriate calling code. The zero at the end of the pragma is the number of words of occam workspace required by the function; since C functions create new SPARC stack frames for their workspace, none needs to be provided in the occam world.

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 that the details of the C mechanism are handled by the C compiler. This function is passed a single parameter, which is the region of the occam workspace containing the occam parameters. This region appears as an array of *`word`*s, where the type *`word`* is a *`union`* of all data types that can be represented in a single word.

For the above example, we need integers (*i*) and pointers to integers (*ip*):

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

A tool (*ocinf*) has been written by another member of the project team (C. S. Lewis) to generate both the interface functions and the occam PRAGMAs automatically from the C prototypes. A demonstration using the *X11* library is provided in the KRoC release.

Because of the endian problem, parameters occupying more or less than a single word have to be converted from one format to the other. Simple 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 procedures are provided for this.

Routines written in other languages, such as Pascal or Fortran, can also 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 interface.

9. Problems and Solutions

9.1. *Endianism*

The transputer is a little-endian machine and the SPARC is big-endian. In the transputer, BYTEs and BOOLs in arrays occupy single bytes, but simple variables of these types are held in words. For parameters passed by reference, the object has to be in the correct (little) end of the word. Hence all accesses to bytes have to be modified (by inverting the low-order two bits of their addresses). This is a small overhead in block moves (generated by move instructions and in/out communications). Later releases of KRoC may eliminate this problem by modifying the occam compiler to allow it to generate code for a (mythical) big-endian transputer.

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 double-word *ldd* and *std* instructions, but they are in the wrong order. However, there is another problem arising from the SPARC requiring 64-bit objects to be aligned on eight-byte addresses, whereas the transputer needs only four-byte alignment. Double-word alignment requires further change to the occam compiler.

9.2. *Floating Point*

Since both the transputer and the SPARC conform to the IEEE 754 floating-point standard [9] (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.

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 reloaded), 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 `fpremstep`) 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.

9.3. *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 possible, 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 pre-condition is not always used;
- conditional jumps (`cj`) – the zero left on the stack when the jump is taken is sometimes never used;

- long arithmetic (`lsum`, `ldiff`, `lmul`, `lshl`, and `lshr`) – the second word is not always required;
- range checks (`csub0` and `ccnt1`) – the value checked is not always used;
- addition is sometimes used for checking (`add` and `adc`) – the result is ignored;
- indexing arrays of timers (`ldl`) – the transputer does not, of course, provide arrays of timers and the same one is used regardless of index. However, that index is always checked for range violation and left on the stack for natural wastage.

In each of these cases, where the stack level cannot be determined unambiguously, the translator records the fact that one or more items 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. (Typically, it happens only once in several hundred instructions.) For this reason, the modified compiler mentioned above provides additional information about the stack level in these cases.

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 a label.

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

Potentially, there is a similar problem with the floating-point stack (for example, in the `fprem` sequence), but it seems never to arise in code generated from occam.

10. Performance

10.1. *Validation*

The 'CG test suite' (developed by INMOS) comprises 27 test programs for occam 2, with a further 18 for occam 2.1. It contains some 7,000 tests consisting of more than 50,000 lines of occam. KRoC passes all of these, with a few small modifications to allow for the different architecture of the SPARC (e.g. `KERNEL.RUN` must be tested with a SPARC binary rather than a transputer one).

However, the CG test suite is a test for compiler correctness and not a test for the correct implementation of transputer instructions, which is what is really required for KRoC. It builds great confidence, but we need to develop a further test suite that checks specifically for the latter. Such a test suite would be helpful in the early stages of retargeting.

10.2. *Process Management Overheads*

We use a simple benchmark* that starts up `N` pairs of processes, transmits `M` messages between each pair, and shuts down all pairs of processes. Each message consists of one integer (four bytes). The test is then re-timed skipping the transmission of messages, but

* The source codes for the benchmarks described in this paper are included in KRoC releases from version 0.7beta onwards.

retaining the inner `SEQ`-loops of each process. Subtracting the former time from the latter leaves us with the overheads for transmitting `N*M` messages – these overheads are primarily those for context-switching (i.e. process synchronization and re-scheduling), plus the cost of the actual *memory-to-memory* transfer of four word-aligned bytes. Continuing the benchmark by removing the inner `SEQ`-loops and, finally, the inner pair of processes lets us get handles on the overheads for managing the `SEQ`-loop (which are high on transputers because of the inefficiency of its *loop-end* instruction) and for process startup and shutdown (measured together).

The benchmark is compiled and translated by KRoC with a flag that makes occam timers yield UNIX *user*-time, rather than *real*-time. UNIX *user*-time reports with a granularity of 10 milliseconds and with an absolute accuracy no better than about 100 milliseconds. Therefore, values need to be set for `N` and `M` in the benchmark that result in timings of the order of 10 seconds and above, so that the relative errors in the timings are small.

Although the number of processes being scheduled has no effect on the number of instructions executed to effect a context switch, memory is involved and so caching effects have an impact. Adjusting the value of `N` (the number of process pairs) changes the workspace requirements of the benchmark. Since the benchmark is repetitive, having a small `N` enables it run almost entirely within cache. For large `N`, the workspace needed is much greater than the cache available and, since the processes are *round-robin* scheduled and hardly anything happens within each process, no advantage can be taken from that cache.

The following tables give results from this benchmark compiled under KRoC *0.7beta* and executing on a 60-MHz SPARC-20 Workstation with 1 Mbyte fast cache and running SunOs 4.1.3. The *External Memory* figures relate to 1,000,000 processes* communicating 256 messages between each of 500,000 pairs. The *Cache Memory* figures are for 100 processes communicating 2,560,000 messages between 50 pairs. For comparison, results are also given for code produced by the Southampton Portable occam Compiler (SPoC) [10], with and without *gcc −O2* optimization, running on the same machine. Finally, results are given for transputer code produced from the standard *Toolset* compiler running on 20-MHz (revision D) T9000 and 20-MHz T800 transputers. (Note: the code and workspace requirement for the 100-process benchmark fits inside the 4 kbytes of on-chip SRAM in a T800 transputer.)

Context-Switch Overheads (microseconds)		
Systems	External Memory	Cache Memory
KRoC	1.06	0.46
SPoC	3.83	1.87
SPoC (*−O2*)	3.16	0.96
Toolset (T9000)	2.29	1.25
Toolset (T800)	2.17	1.36

* Workspace limits imposed by a UNIX system on user stack space may prevent the running of this benchmark with 1,000,000 processes. Under KRoC, each process in this benchmark requires 32 bytes of workspace; under SPoC, somewhat more is needed.

Startup+Shutdown Overheads (microseconds)		
Systems	External Memory	Cache Memory
KRoC	1.60	0.70
SPoC	18.5	8.44
SPoC (−O2)	16.4	4.86
Toolset (T9000)	4.42	1.84
Toolset (T800)	4.22	2.10

SEQ Loop Overheads (microseconds)		
Systems	External Memory	Cache Memory
KRoC	0.20	0.20
SPoC	0.14	0.14
SPoC (−O2)	0.04	0.04
Toolset (T9000)	0.33	0.33
Toolset (T800)	1.53	0.65

The context-switch overheads include the time to move the message (four word-aligned bytes) between the process workspaces. The *startup+shutdown* overheads give the combined time to start up *and* shut down a process – separate timings are hard to measure. Note also that process shutdown involves a barrier synchronization.

KRoC process-management overheads are about three times lower than those for 20-MHz transputers (T8 or T9), which is (the least) we should be looking for from a processor that is clocked at three times the speed (60-MHz SPARC). The SPoC context-switch figures are slightly greater than those for the transputers, but the startup and shutdown overheads are significantly greater (which will affect the freedom with which low-level PAR constructs can be applied). KRoC suffers from the *loop-end* difficulty it inherits from the transputer.

10.3. *Serial Floating-Point Performance*

No matter how fast multi-processing can be made, it is important that this is not gained at the expense of other fundamental parameters, such as floating-point performance.

Daxpy is a low-level component 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)
:
```

This is a severe test for modern processors as it enables practically no advantage to be taken of cached memory (assuming the array sizes are large). Nevertheless, the test reflects per-processor MFLOPS commonly sustained by *state-of-the-art* HPC facilities on real applications (which generally range from 2% to 30% of their theoretical peak [11, 12]).

As with the previous benchmark, versions were run to show its performance in *external* and *cache* memory. For the former, the array sizes were set to 100,000 and the computation repeated 100 times. For the latter, these numbers were reversed. All results were checked for numerical accuracy. The same SPARC and transputer platforms as reported in the previous section were used for these tests.

Daxpy Performance (MFLOP/second)		
Systems	External Memory	Cache Memory
KRoC*	2.7	3.6
KRoC unrolled*	4.0	6.3
KRoC + GUY	4.8	8.6
KRoC + as (*little-endian*)	5.4	12.3
KRoC + C (-O)	6.1	14.0
KRoC + as (*big-endian*)	6.8	16.0
SPoC*	1.2	1.3
SPoC unrolled*	3.9	5.4
SPoC (-O2)*	3.7	5.6
SPoC unrolled (-O2)*	6.0	14.2
C	3.1	4.4
C (-O2)	5.8	14.4
Toolset (T9000)*	1.4	1.9
Toolset unrolled (T9000)*	1.9	3.2
Toolset + GUY (T9000)	1.7	2.2
Toolset (T800)*	0.45	0.52
Toolset unrolled (T800)*	0.58	0.70
Toolset + GUY (T800)	0.75	0.96

Several different versions of daxpy are reported in the above table. For KRoC, the first row gives the (double-precision) floating-point performance for the raw occam code given above. The next row has the loop unrolled 16 times, which gives significant run-time savings on array element address computations and on array index range checks. The third row has the unrolled loop replaced by in-line transputer assembly language (GUY). The fourth row has occam calling the daxpy routine written in SPARC assembly language (*as*), but with the data still in transputer *little-endian* format (which means no advantage can be taken of double-word loads and stores). The next row shows the result of occam calling C (separately compiled with *gcc -O*). Note that the data has to be organized into *big-endian* format for this to work. The last row in this first section shows the result of hand-crafted SPARC assembly language that also takes advantage of the data being in *big-endian* format.

The next section of the table shows the performance of SPoC on the original occam code and its loop-unrolled form. The third section repeats this but with *gcc* optimization on during SPoC compilation. The fourth section shows the performance of an all-C version of *daxpy*, with and without *gcc* optimization and, of course, with no range checks.

The final two sections give the results of the original occam, the unrolled occam, and the occam plus GUY code on T9000 and T800 transputers.

From occam, both KRoC and SPoC give serial floating-point performance as good as *gcc* from C. However, KRoC needs to drop into native assembly language, or separately complied and optimized C, to match *gcc* with optimization. SPoC is the clear winner here since it can directly exploit *gcc* optimization in its second stage (having first compiled occam into C).

* Includes range-checking of array indices.

10.4. *Discussion*

Putting the figures from the previous two sub-sections together, it is relevant to express process management overheads in terms of *obtainable* serial floating-point performance (as opposed to some mythical *peak* figure). From Section 10.2, this best performance varies between about 6 and 15 MFLOP/second, depending upon how successfully we can hit the cache. We can now generate the following tables:

Context-Switch Overheads (FLOPs)		
Systems	External Memory	Cache Memory
KRoC	6.3	6.9
SPoC	22.9	27.9
SPoC (`-O2`)	18.9	14.3
Toolset (T9000)	4.3	4.0
Toolset (T800)	1.6	1.3

Startup+Shutdown Overheads (FLOPs)		
Systems	External Memory	Cache Memory
KRoC	9.6	10.4
SPoC	110.8	126.0
SPoC (`-O2`)	98.2	72.5
Toolset (T9000)	8.4	5.9
Toolset (T800)	3.2	2.0

For floating-point intensive applications, we can now make decisions about how much parallelism (or *multi-threading*) it is safe to allow ourselves before their overheads become noticeable. Note that these decisions are now independent from considerations of cache hit-rates (the columns contain very similar costs).

A most striking observation from the above tables concerns the extraordinary capabilities of the T800 transputer. The T800 can start up a process, shut down a process, or switch context at almost the same speed with which it can perform (64-bit) floating-point arithmetic! Combine that with the recollection that the T800, when it was first available commercially, was the fastest floating-point microprocessor in the world and we have an achievement from which *state-of-the-art* architectures have retreated dramatically. That achievement needs regaining for two reasons. Firstly, it enables the direct implementation of fine-grained parallel design (which is the starting point for most methodologies for modelling the natural world). Direct implementation implies simplicity and, hence, greater confidence, verifyability, and security. Secondly, it is a prerequisite for low-latency startup for external communications, which is itself a prerequisite for scalable performance on multi-processor platforms (private memory, shared memory, or virtual shared memory).

The T9000 transputer, clocked at the same speed as the T800, has improved its floating-point performance (currently by a factor of three and a bit*), but has only slightly improved its process management – hence, the increased costs in the above tables. KRoC, applied to a modern RISC processor (a 60-MHz SPARC-20), brings that processor into line with the T9000, yielding process costs in terms of arithmetic that are almost as low, without

* Note that the in-line `GUY` code, optimized for the T800, is not helpful for the T9000.

damaging the speed of that arithmetic. SPoC is three to ten times more expensive than KRoC, depending on how much *low-level* parallelism is present. Nevertheless, all these systems have costs that are sufficiently low to qualify as *virtual transputers*, enabling us to retrieve most of the scientific and engineering achievements of the original T800.

11. Future Work

Work is in progress on a number of extensions to the present KRoC, as well as on versions for other architectures (see Section 1).

11.1. *Multi-Processor Systems*

A major extension will be to run occam programs distributed over a network of workstations or tightly coupled multi-processor systems. The system under development is based on the design of the Virtual Channel Processor (VCP) of the T9000 transputer, and will provide support for virtual channels using whatever communications fabric (TCP/IP ethernet, DS-links, shared memory, ...) is to hand. A side-effect of this will be an occam channel interface to UNIX sockets and, for example, the ability to start experimenting with occam multi-threaded *World Wide Web* servers ...

11.2. *Calling C*

The current mechanism provides a conventional *procedure* interface. It would be much better to provide a *channel* interface to enable occam to call C functions without blocking and to receive call-backs in a natural way.

11.3. *File Access*

At present, KRoC can only access files through redirecting the standard UNIX streams `stdin`, `stdout`, and `stderr`. Later versions of KRoC will support the *Toolset* `SP` interface as an alternative.

11.4. *Optimization*

KRoC relies on the *Toolset* compiler to generate transputer code and, currently, this contains no optimization stage. The retargeted code, therefore, not only has to accept the overhead of not exploiting its full register set for serial computation*, but it also has to be derived from transputer code that is non-optimal even for transputers.

Given these constraints, it is interesting that KRoC code from pure (but *unrolled*) occam is between 45% (*in-cache*) and 70% (*ex-cache*) of the performance of `gcc`, with full optimization set, from C on code such as *daxpy* (and is faster than unoptimized `gcc`). An optimizing occam *Toolset* compiler does exist, but this has not been released by SGS-Thomson Microelectronics. In the meanwhile, inner loops of computationally intensive processes can be coded in native assembly language or C (separately compiled and optimized using standard tools) – a practice long familiar for HPC applications programmed in traditional high-level languages.

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

* This is, of course, a positive benefit for parallel computation ...

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 does not follow.

Another possibility is for the translator to keep a record of the values of constants, 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 somewhat 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.

11.5. *Debugging*

When faced with a component that doesn't work properly, a bad engineer tries to find out what's wrong with it. A good engineer wants to know what's right with it – what evidence is there that it has been implemented to meet its specification? The good engineer looks at the implementation and tries to understand how it does its task. Most faults in a well-engineered system will not stand close inspection. Faults in a badly-engineered system are not worth wasting time on – better to throw it away and re-design properly. Occasionally, subtle faults in a well-engineered design will escape our attention – the implementation looks good but it still doesn't work. Only then, *as a last resort*, do we switch the component on, attach probes, and watch it in action (i.e. start 'debugging').

By virtue of the engineering within the language [13, 14], a vast range of common and subtle errors (e.g. aliasing problems, race conditions, data and message structure violations, syntactic/semantic inconsistencies, unexpected type-coercion or loop-exit or operator-precedence, ...) simply cannot occur in occam systems. Therefore, the *need* for debugging tools is greatly reduced and they are not a high priority in *occam-for-all*.

Nevertheless, we have resisted the temptation to print the ancient UNIX response ('Eh?') upon hitting a run-time error, and do, at least, say what kind of error (range-violation, divide-by-zero, ...) has occurred. Future releases will relate this to the source line-number and file-name of the offending code. It should be possible, with some effort, to port (or reconstruct) the *post-mortem analyser* `idebug`, but we are waiting to see the level of demand. An interactive *vivisector* (capable of setting breakpoints, changing variables, step-executing processes, ...) is considered to be of only marginal use and liable to encourage bad engineering practice.

11.6. *Priority Scheduling*

The only feature of occam not supported is `PRI PAR`, which is treated as `PAR`. Handling priorities is relevant for ensuring that external communications are started as early as possible. This allows the exploitation of any capability offered by the underlying hardware and software for overlapping communications and computation. Priority Scheduling is also needed in real-time applications to allow high computational loadings to be imposed on processors and still provide guarantees of meeting real-time deadlines [15].

Using ideas discussed in [15], we are investigating a full implementation of `PRI PAR` – i.e. one that allows *any* number of components (current occam implementations limit this to *two*). This will involve significant changes to the *Toolset* compiler and to the KRoC kernel. Nothing will be put in that damages the current process management overheads.

11.7. *Language Development*

After more than a decade of use, the occam programming language has proven remarkably mature and robust for the design and implementation of efficient multi-processing applications. It offers a level of security and lightness in overhead that remain unmatched by current alternatives.

However, despite its achieving widespread industrial and academic acceptance, anecdotal evidence suggests that this base is diminishing. The main reasons given are perceptions that occam is a specialist language for the transputer and that it is not (built upon) C or Fortran. The former is being addressed by both SPoC and KRoC – projects that should, and technically could, have been started ten years ago. There is not much that can be done about the latter ... except to question continually the wisdom of accepting engineering methods that allow errors to be made that are preventable with modern tools (and to consider our legal liabilities concerning such acceptance).

No tool provides universal capability and occam was never intended to be the last word in parallel processing! In one sense, occam is like an assembly language for parallelism. It provides `PAR` and `ALT` constructs and communication primitives, but allows these to be combined to create *any* pattern of process interaction – including some that may deadlock or livelock. Now that we have more experience, it may be time to consider designing higher level languages that only permit synchronization regimes that can be guaranteed free from deadlock or livelock (see [16, 17]).

On the other hand, occam is sometimes too conservative and unnecessarily limits the amount of parallelism available to an application. The chief problem here is its attitude to *sharing*. Data that is shared between processes is frozen (i.e. read-only) for the duration of that sharing; processes only interact through explicit one-to-one communication channels. This achieves the vital benefit of eliminating accidental race hazards (non-determinism), an achievement that is not addressed by the available alternatives. However, it is possible to *weaken* the rules on sharing (in ways that will allow read/write permissions on shared data to be more dynamic) and still retain all the previous security. High level concepts to capture these rules may need to be introduced in the language to make this safe. These issues will become especially important in the context of shared-memory (or virtual-shared-memory) multi-processors.

During the period 1989–92, INMOS developed and published proposals for an extended language called occam3 [18], which included mechanisms for securely `SHARED` channels, `CALL` channels, and *programming-in-the-large*. Only about 10% of these extensions have been implemented in the occam2.1 language (which KRoC has inherited from the SGS-Thomson Microelectronics compiler). The remaining 90% need some serious study into implementation strategy.

12. Conclusions

The approach to retargeting occam proposed here has been shown to be relatively easy to implement, and to produce code of acceptable efficiency. Those requiring top performance can drop into C or assembly language for serial inner loops. Despite the obvious differences in generating code for different architectures, it is proving easy to retarget to several machines other than the original SPARC. The original goal that the system should work with an unmodified occam compiler is satisfied only for programs compiled as a single unit, but is no longer important now that we can release our own version of the compiler.

Despite having to use a software (rather than microcoded) kernel, the results in Section 10 show that key properties of the transputer do survive this retargeting. Context-switching on a 60-MHz SPARC-20 ranges between a half and one microsecond (depending upon cache hits). Process startup and shutdown (combined) costs range from three-quarters to one-and-a-half microseconds.

These are somewhat lighter than the overheads incurred by most other systems for process management on such architectures and raise the prospect of a new use for occam as a *very-lightweight-threads* mechanism for a (uni-processor) workstation. Current lightweight threads libraries require great care from the programmer, as incorrect usage is not machine checked. In occam, it is simply not possible to write a procedure that is 'thread-unsafe' and use it as part of a threads application.

The current results for KRoC on a single processor are a necessary condition for scalable performance on multi-processors. Work is in hand to extend the KRoC kernel to support multi-processor platforms in a way that is retargetable to different communications fabric. If all goes well, the result will be to enable occam applications to be configurable to a wide range of parallel architectures. On such 'virtual transputer systems', occam (or its descendants) has one further benefit to confer: the efficient reconciliation of *application-specific* parallel software with *general-purpose* parallel hardware, where the software and hardware parallel structures have no correspondence [19].

KRoC binary releases are freely available from [2]. Source releases of the translator and kernel are available for academic research (e.g. further retargeting) through contacting the authors. KRoC is being used at the University of Kent and elsewhere for supporting occam teaching and parallel computing generally. Teaching materials are included with the binary releases. We welcome and need feedback from all who experiment with this system.

13. Acknowledgements

We would like to thank all the members of the *occam-for-all* team at Kent for ideas and encouragement. In particular: Michael Poole, whose implementation of occam for the PC [20] proved that this sort of thing was possible, and who modified the occam compiler as described in Section 8 and ported the KRoC kernel to the DEC Alpha; Carl Lewis, who retargeted the KRoC translator to the DEC Alpha and who wrote the tool for generating C interfaces; Kevin Vella, who is investigating the multi-processor implementation for KRoC; Vedat Demiralp, for repeatedly running the test suite on each new version, and who is also working on multi-processors; and Dave Beckett, for writing the KRoC driver, controlling the ever-increasing numbers of different versions, and setting up and managing the public releases.

We also thank Kneale Rothwell, Graham Shaw, Keith Wollacott, and their supervisor Andrew Smith, whose student project at Kent produced the first SPARC kernel [21], from which the KRoC version has evolved.

We are especially grateful to SGS-Thomson Microelectronics, both for making their compiler sources and test-suites available to this project and for technical advice.

Finally, we also thank Denis Nicole, of the University of Southampton, who first seriously proposed this approach at the occam Porting Workshop held at the University of Kent in September, 1992, and who estimated it would be about two weeks' work.

References

[1] *occam-for-all*. See <URL:http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/>.

[2] KRoC release information. See <URL:http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/kroc/>.

[3] SGS-Thomson Microelectronics Limited. occam 2.1 Language Reference Manual. See <URL:http://www.hensa.ac.uk/parallel/occam/documentation/>.

[4] INMOS Limited. The transputer instruction set – a compiler writers' guide. Prentice Hall, 1988. ISBN 0–13–929100–8.

[5] D. A. P. Mitchell, J. A. Thompson, G. A. Manson, and G. R. Brookes. Inside the Transputer. Blackwell Scientific Publications, 1990. ISBN 0–632–01689–2.

[6] M. D. May, P. W. Thompson, and P. H. Welch (eds). Networks, Routers, and Transputers. IOS Press, 1993. ISBN 90–5199–129–0.

[7] The T9000 Transputer Instruction Set Manual. INMOS Limited, 1993.

[8] SPARC International. The SPARC Architecture Manual. Prentice Hall, 1992. ISBN 0–13–825001–4.

[9] ANSI/IEEE Std 754–1985, IEEE Standard for Binary Floating-Point Arithmetic. The Institute of Electrical and Electronic Engineers, Inc, 1985

[10] M. Debbage, M. Hill, S. Wykes and D. Nicole. Southampton's Portable occam Compiler. In *Proceedings of WoTUG–17: Progress in Transputer and* occam *Research*, edited by R. Miles and A. Chalmers. IOS Press, April, 1994. ISBN 90–5199–163–0

[11] S. Saini and D. H. Bailey. NAS Parallel Benchmarks Results 3–95. Report NAS–95–011, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Mail Stop T 27A–1, Moffett Field, CA 94035, USA.

[12] Crisis in HPC Workshop. See <URL:http://www.hensa.ac.uk/parallel/groups/selhpc/crisis/>.

[13] C. A. R. Hoare: The Transputer and occam; a Personal Story. *Concurrency: Practice and Experience*, vol.3(4), pp. 249–264. August, 1991.

[14] P. H. Welch: The Role and Future of occam. In *Transputer Applications – Progress and Reports*, edited by M. Jane, R. Fawcett, and T. Mawby. ISBN 90–5199–0790. IOS Press, March, 1992.

[15] E. Ploeg, J. P. E. Sunter, A. W. .P. Bakkers, and H. W. Roebbers: Dedicated Multi-Priority Scheduling. In *Proceedings of WoTUG–17: Progress in Transputer and Occam Research*, pp 18–31, edited by R. Miles and A. Chalmers. ISBN 90–5199–163–0. IOS Press, April, 1994.

[16] P. H. Welch, G. Justo, and C. Willcock: High-Level Paradigms for Deadlock-free High-Performance Systems. In *Transputer Applications and Systems '93*, pp 981–1004, edited by R. Grebe et al. ISBN 90–5199–140–1. IOS Press, September, 1993.

[17] J. Martin, I. East, and S. Jassim: Design Rules for Deadlock Freedom. *Transputer Communications*, vol.2(3), pp. 121–133. September, 1994.

[18] G. Barrett: occam 3 Draft Language Reference Manual. See <URL:http://www.hensa.ac.uk/parallel/occam/documentation/>.

[19] P. H. Welch: Parallel Hardware and Parallel Software: a Reconciliation. In *Proceedings of the ZEUS'95 & NTUG'95 Conference*, Linkoping, Sweden, pp 287–301, edited by P. Fritzson and L. Finmo. ISBN 90–5199–22–7. IOS Press, May, 1995.

[20] M. D. Poole. An Implementation of occam 2 Targetted to 80386, etc. WoTUG News No 18, 1993.

[21] K. Rothwell, G. Shaw, K. Wollacott, and A. Smith. Porting the INMOS occam Compiler to the SPARC Architecture. In *Proceedings of WoTUG–18: Transputer and* occam *Developments*, edited by P. Nixon. ISBN 90–5199–222–x. IOS Press, April, 1995.