Communicating Process Architectures 2001
Alan Chalmers, Majid Mirmehdi and Henk Muller (Eds.)
IOS Press, 2001

tranx86 – an Optimising ETC to IA32 Translator

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

Abstract. This paper describes `tranx86`, a program which converts Extended Transputer Code (ETC) from a modified Inmos occam compiler, into IA32 code for execution on the Intel i386 family of processors within the KRoC/Linux system. Several optimisations are employed in an attempt to maximise performance on this family of processors, including optimisations in the CCSP run-time kernel. These include a graph-colouring type register allocation scheme and various inlining of code. While `tranx86` is mostly architecture dependent, effort has been made to allow the use of arbitrary schedulers, although currently CCSP is the only fully supported one.

Various benchmark programs are used to compare the performance of this translator with the old system, giving significant time wins in some cases. For the `commstime` benchmark program on an 800 MHz Pentium-3, the old KRoC/Linux system gave 233 ns per communication (2 context switches); the new one, with optimisations and inlining, gives 67 ns per communication – more than a 3-fold reduction in overheads.

1 Introduction and Motivation

The KRoC[1] system, on the Linux/i386 platform, is composed of a number of smaller programs which are used to produce executables for the target platform (in this case the IA32[2] – Intel Architecture 32-bit – platform[1]) from occam[3] sources. A modified Inmos occam compiler (`occ21`) is used to generate Extended Transputer Code (ETC) [4] from occam sources. ETC can be thought of as *Virtual Transputer Byte-Code*. A translator is then used to turn ETC into native i386 code, which is then linked with libraries and the CCSP [5] run-time kernel to produce an executable. Figure 1 shows this layout, with routes for two different translators, `tranpc` and `tranx86`.

The original translator used in the KRoC/Linux system was `tranpc`. This was written by Michael Poole in occam and, to a small extent, based on the `octran` translator from the SPARC port of the KRoC system. `tranx86` was a complete re-write, this time in C, incorporating a mixture of ideas from `tranpc` and `octran`. The structure of `tranx86` and `tranpc` are quite different. `tranpc` took a very direct approach, generating machine-code bytes directly from the ETC. While this worked, developing from it proved to be fairly difficult, thus a new one emerged.

One obvious problem with translating from one architecture to another, in this case from ETC to IA32, is the potential loss of efficiency, since the target architecture might have a largely incompatible instruction set. ETC is effectively code for a *virtual transputer*, a stack machine with instructions for CSP [6, 7, 8] concurrency operations. The target instruction set (IA32) is a CISC[2] architecture (on the outside only in the Pentium-Pro and above), and is register based. Fortunately the floating-point co-processor on the IA32 can be treated

[1] more commonly known as the i386 platform
[2] Complex Instruction Set Computing

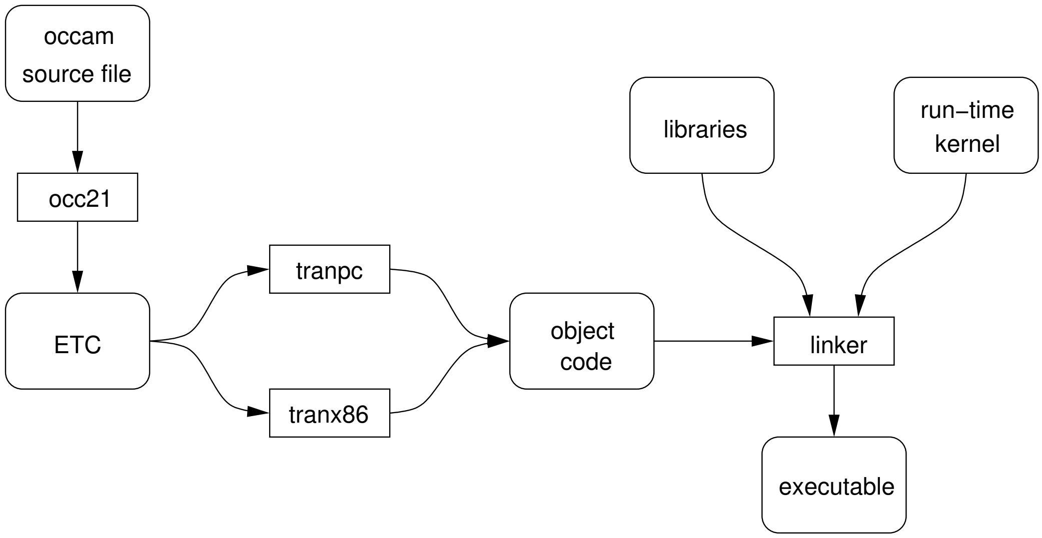

Figure 1: Stages in turning occam sources into i386 executables in the KRoC/Linux system

as a stack machine, which makes the translation of floating-point code relatively painless, although implementing a register-based approach would not be too difficult.

The majority of existing KRoC translators are for RISC architectures, `octran` and `oc-X` [9] for example. Although `tranx86` generates CISC code, existing optimisations have been reused where possible.

1.1 The virtual transputer architecture

The virtual transputer targetted by the occam compiler is a stack machine, similar to the T800 transputer. It has a 3-level integer stack and a 3-level floating-point stack. A special register exists which provides the workspace address of the current process, 'Wptr'. The instruction set is based on a combination of T800 and T9000 transputer instructions [10] (T800 instructions form a subset of T9000 instructions). This instruction set is complemented by various additional instructions, for example instructions for dynamic memory allocation and mobile communication [11]. Figure 2 shows the register layout of the virtual transputer. In addition to the workspace-pointer 'Wptr', there is also the instruction pointer 'Iptr' and three queue pointers which are used to manage the timer-queue and run-queue. There are also some status registers which are not shown.

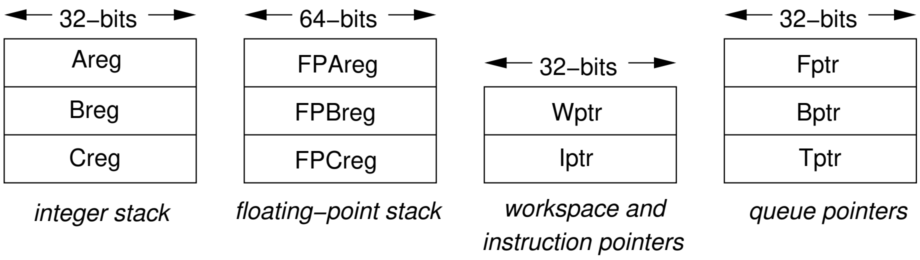

Figure 2: Registers in the virtual transputer

1.2 The IA32 architecture

The IA32 architecture is primarily a register based CISC architecture. There are four general-purpose 32-bit integer registers, stack and frame pointer registers, index registers, and various segment registers. The CISC nature of the IA32 allows complex instruction encodings to be used, which greatly enhances the expressive power of the instruction set. Figure 3 shows the register layout of the IA32 architecture. The layout of the FPU (floating-point unit) registers shows the register view, whereas the translator treats it as a stack machine. When treated like a stack, a top-of-stack pointer (held in part of the FPU control word) indexes one of the FPU data registers ($R0..R7$). The segment-registers are not used at all, although the FS and GS registers could be utilised (generally they are only used by debuggers). The translator uses the four general-purpose and base-pointer (EBP) registers most of the time, and uses the stack-pointer (ESP) and index registers some of the time.

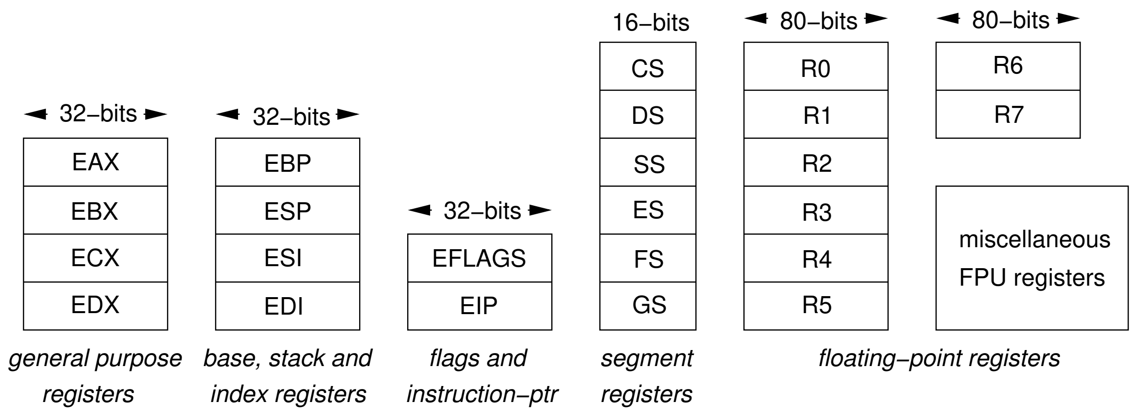

Figure 3: Registers in IA32 processors

2 The Structure of tranx86

Figure 4 shows the overall organisation of `tranx86`, starting from the ETC output from the compiler. Sections 2.1 through 2.8 describe the various functional components.

Internally, the code of an occam program is held on one of two *chains*. Chains in this context are very much like linked-lists, except that a lot of cross-references come into existence along the way. The first chain is created as the (binary) ETC[4] file is read. After pre-optimisation, this is either passed to the translator or output in a textual format. Having textual output at this point provides a good method for examining the output of the occam compiler.

The second chain is generated by the translator (section 2.2) and consists of a sequence of *intermediate-code* (IMC) blocks. These blocks break the program up into fairly large chunks, such as program code, program data, global entry-points, etc., each with its own particular additional data. On blocks marked as 'code' hangs a list of instructions which represent the translated program. Figure 5 shows an example of a simple IMC chain.

The instruction mnemonics used in the intermediate code relate largely one-to-one with IA32 instructions. This arose from the specialist nature of certain instructions, such as '`movzbl`', which zero extends an 8-bit quantity to a 32-bit quantity during the move. In some ways, this restricts the ability to target multiple (differing) instruction sets. Work is in progress to replace the more specialist instructions with sequences of simpler ones, leaving

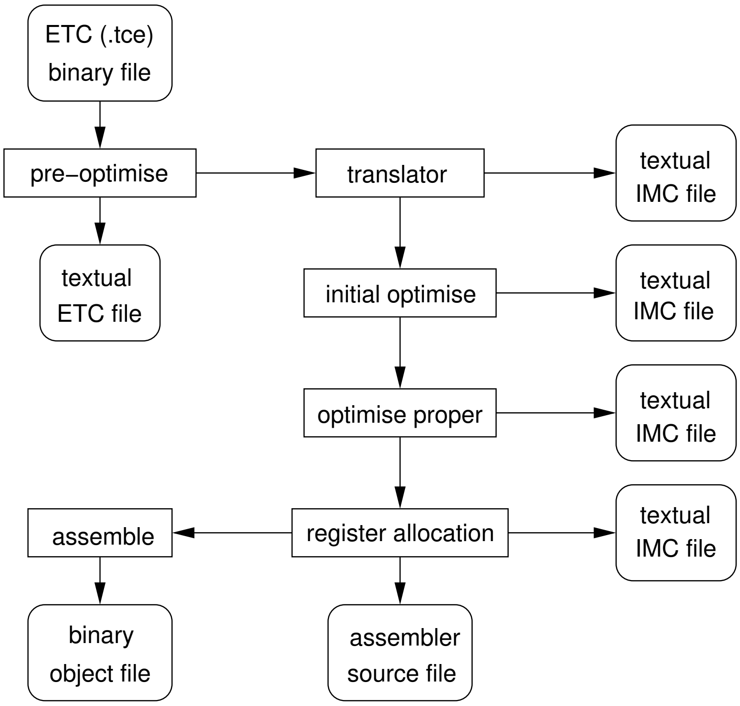

Figure 4: Structure of `tranx86`

generation of the replaced instructions to the assembler-generating stage. On the more positive side however, the encoding of arguments differs significantly compared with IA32 arguments. We split the arguments into two distinct groups: *input* operands and *output* operands. Operands can also take a variety of flags, one of which indicates an *implied* operand – for instructions which use registers not present in the encoded arguments.

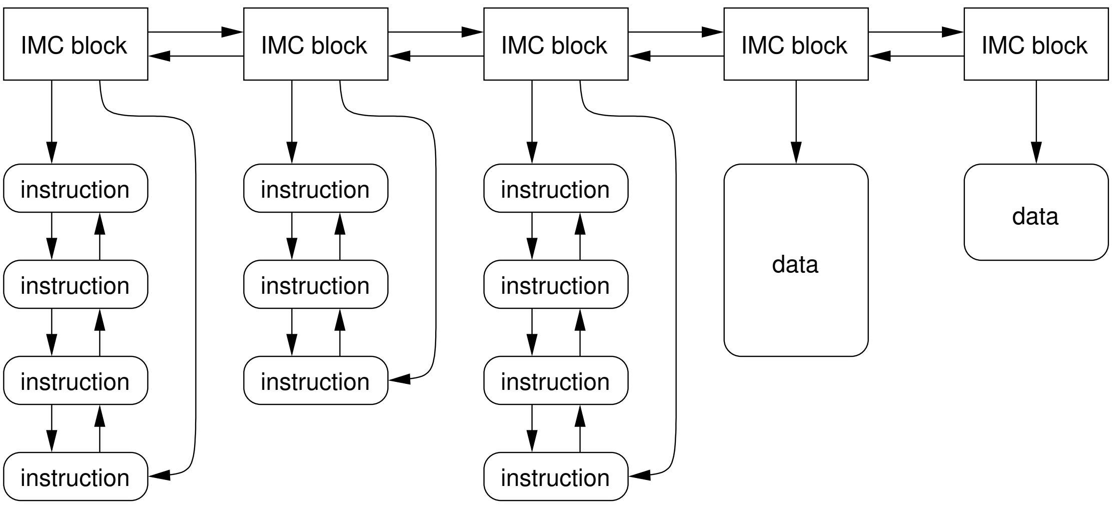

Figure 5: Internal program representation in `tranx86`

Instead of inventing our own intermediate code representation, we could have used `gcc`'s RTL (register transfer language) [12], which has a well established following, and used the `gcc` back-end to generate the native code. However, it is envisaged that `tranx86` will be used to generate code for targets which `gcc` does not support (the Intel IXP-1200 network processor was discussed at one point). In these cases, re-targetting `tranx86` is likely to be much easier than re-targetting gcc. The possibility of using gcc has not been removed altogether, merely deferred for the future.

2.1 Pre-optimisation

The pre-optimiser performs some simple transformations on the ETC code to make the task of translation easier. One feature of KRoC is that the occam compiler is used to generate code for a virtual transputer with T800 characteristics. This includes a particularly tricky floating-point remainder (modulo) operation (with FPREMFIRST and FPREMSTEP), which needs to be transformed into a more conventional remainder operation, in this case the corresponding instruction for the T9000 transputer (FPREM).

If CCSP has been selected as the target run-time kernel, channel communications of 1 and 4 bytes are turned into communication instructions specific to those lengths. While only being a minor thing, this allows a significant improvement in the handling of BYTE and INT channel communications – mostly because the inputting and outputting processes no longer need to specify the length of the data being communicated.

2.2 Translation

The translator moves along the ETC chain instruction by instruction, putting translated code onto the IMC chain as it goes. Since ETC is targetted at a *stack-machine*, a mapping between the virtual transputer stack and virtual target registers is maintained, along with information such as stack depth, floating-point stack depth and any previous stacks (stacks can become nested). A new virtual register is created each time the virtual transputer stack is pushed, and is forgotten when popped from the stack. As well as this stack state information, which is present in tranpc and octran, a mapping between virtual registers and any related constants is maintained (constant-map).

The virtual-transputer workspace pointer 'Wptr' is mapped into the IA32 'EBP' (base-pointer) register. This is hidden though through the use of a special virtual register called REG_WPTR. Along with this are REG_JPTR and REG_LPTR which are mapped into 'ESI' and 'EDI' respectively. The actual fixing of these registers happens during register allocation (section 2.7). If it is desired that these registers should be mapped differently, all that is required is a small change in the register allocator. Some IA32 instructions use registers implicitly (notably the various divide instructions) which is a potential cause of problems. The handling of this is done by inserting special constraint instructions, which link a virtual register to an actual target register. The register allocator performs the fixing of these constraints, as well as resolving any conflicts which may arise from their use.

The target instruction set is CISC, as opposed to RISC, which allows the majority of instructions to take a variety of argument types, including registers, constants, indirect addresses and more exotic variants. The majority of occam variables live in the workspace (as referenced by 'Wptr') which are loaded and stored mostly through the use instructions which use 'Wptr' as a base and provide a constant offset. When constants, local-variables and addresses are loaded into the virtual transputer stack, a corresponding entry in the constant-map is generated which indicates what was loaded. When translating certain other instructions (especially those that pop the stack), the constant map is checked to see if what was loaded onto the stack can be used directly.

2.3 Interaction with the kernel

Interaction with the run-time kernel, which implements the virtual transputer for communication and scheduling, is done through the use of kernel calls[3]. Support for two existing kernels is provided, the original CCSP [5] and a heavily modified version of CCSP which provides additional kernel calls and supports different calling conventions. Parts of the framework are also in place to support MESH[4] [13] and kernel-level schedulers such as *libcsp* [14] or pthreads.

The original combination of `tranpc` and CCSP used a push-pop method (on the C stack) to pass arguments from the translated program to CCSP. When `tranpc` encountered something requiring a kernel call (process scheduling and communication instructions), it pushed the virtual transputer stack onto the C stack then called the relevant entry-point from a lookup table (pointed at by the 'ESI' register). Figure 6 shows this calling sequence. Although this works, it is non optimal – arguments are pushed onto the stack only to be removed again after the jump.

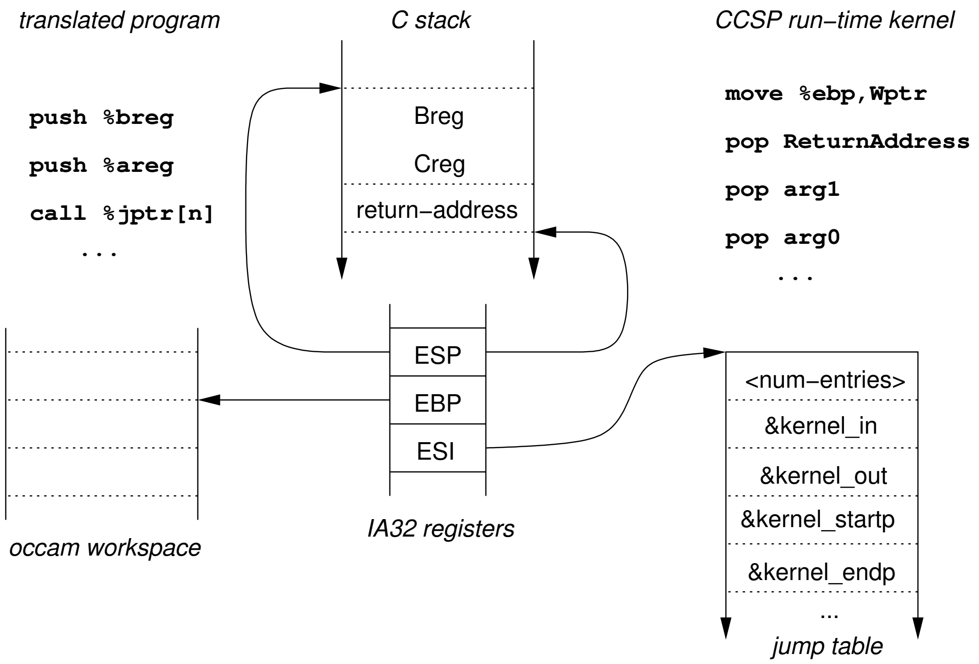

Figure 6: Original run-time kernel calling sequence

The enhanced kernel-call interface for the modified CCSP allows for a variety of calling conventions which do not require the use of the 'ESI' register. A table of kernel entry-points is held in the translator which holds a named label for each entry-point, along with how to to pass arguments and call it. For argument passing, parameters can either be pushed onto the C stack (old behavior), or the relevant virtual registers can be constrained to specific IA32 registers. The method used depends on how the modified CCSP handles the other half of the call in C. For the constraint method, the entry-point in the C world (handled by inline assembly macros) arranges for certain variables to be mapped against the registers used. This requires the use of certain features in the GNU CC compiler `gcc`.

[3] our kernel calls (to a user-space kernel) differ significantly from traditional kernel-space kernel calls, which involve privilege-level changes and page-table manipulations

[4] The user-level schedulers in both CCSP and MESH were originally the same, developed by James Moores and Marcel Boosten.

For the actual jump to the run-time kernel, the modified interface supports four different methods, listed in table 1. When an occam process is not being run, it uses workspace offsets beneath zero to store run-time state information. How much state is required depends on what the process interacts with. For non-running processes, the return-address is held at offset -1. The "storeip-jump" kernel call takes advantage of this by saving the return address in the workspace before making the kernel call. The "jump" kernel call is used for calls which never return. This includes the ENDP (end-process) call and various run-time error handling calls. Arguments being returned from the kernel to the program are handled in the same way as the arguments passed to the kernel. As it happens, gcc seems particularly reluctant to allow register constraints in the places where we want them – only a handful of kernel-calls return things, and they do this using the C stack.

Method	Mechanics
call	regular call (IA32 call instruction), return-address is left on the stack
storeip-jump	place return-address at 'Wptr[-1]' and jump to the entry-point
regip-jump	place return-address in a register and jump to the entry-point
jump	ump directly to the entry-point (return-address is lost)

Table 1: Different methods of entering the run-time kernel

In addition to the kernel interface enhancements, the 'ESI' register is freed up. The translator generates references to entry-point *names*, and lets the linker resolve the references.

2.4 Run-time errors

The *post-mortem* debugging implemented in the older KRoC/Linux system [15] has been re-implemented in tranx86 along with various deficiency fixes and additions, notably support for floating-point debugging. The floating-point unit runs asynchronously to the integer unit in IA32 processors, so the location at which an error is reported might not be the same as where it originated. Care is taken in the implementation to ensure this is handled correctly.

For more extensive debugging needs, an *execution-trace* functionality has been added. This records the execution position at each point a new source-line instruction (generated by the occam compiler) is encountered. The source position (and other associated information) is stored in special memory locations inside the run-time kernel. When the program exits due to error, this information is printed. Which not much use for ordinary errors (which have their own debugging information) this is useful for diagnosing errors such as segmentation-faults (from invalid memory accesses), which can arise when *transputer-assembly* is inlined into occam source-code. For memory violations in linked C code [16] the reported position will be the point in the occam program at which the external call was made – a C debugger (such as gdb) can then be used to diagnose further.

2.5 Initial optimisations

The first series of transformations are used to clean-up the output from the translator. The first of these is dead-code removal, which is done by removing any code between unconditional jumps and the following label. As well as generating code, the occam compiler also generates a reasonable amount of constant data. This is used to encode constants from the source code. Any constant data considered relocatable, i.e. it is preceded by a label, is moved to the end of the IMC chain. When the output is generated, this will end up in the '.data' segment, and be cached as such by the CPU.

The next transformation finds occurrences of labels as arguments and replaces them with a link to the corresponding `SETLABEL` instruction. This makes it easier to re-arrange code involving labels, since each label knows where its references are.

2.6 Optimisations proper

The bulk of optimisations in `tranx86` deal with re-arranging common sequences of code into shorter forms. Rather than being generic, these optimisations are based on finding certain sequences of instructions and replacing them with shorter or more efficient sequences. If an optimisation has to choose between time or space efficiency, `tranx86` will generate for time efficiency. Modern PCs, which are the primary target here, generally are not short of memory. Figure 7 shows a selection of the transformations performed by `tranx86`.

Original code	Optimised code

`pop %r`
`...`
`move %r, <x>`
	`pop <x>`

`move <x>, %r` `or %r, %r`	`cmpl $0, <x>`
`add $N, %r, %r` `add $M, %r, %r`	`add $(N+M), %r, %r`
`push <x>` `ret`	`jump *<x>`

`cjump CC, Lx`
`jmp <y>`
`Lx:`
	`cjump CC^1, <y>`

`or %r, %r, %r`
`setcc _Z_, %r`
`and $1, %r, %r`
	`xor $1, %r, %r`
	`and $1, %r, %r`

Figure 7: Selection of code transformations in `tranx86`

A large proportion of overhead reduction (compared to the old system) has arisen from the use of a colouring register allocator and constant propagation. A reasonable number of IA32 instructions have their input or output operands fixed to certain registers (often due to limitations in the encoding). The old translator (`tranpc`) handled these by re-writing the registers in already generated code (and inserting `move` instructions if necessary) if there was a register collision. Although this works, it isn't terribly nice.

The constant map helps reduce the amount of code generated by subsituting constants for registers if possible. Constants here also include workspace variables at constant offsets from the workspace pointer. For cases where a constant is loaded then used once, the constant will be subsituted, leaving just the initial load instruction. These are swept away by the optimiser – the register is only used in one instruction. Virtual registers help here since the register is only used just the once – ever! Code involving physical registers is unlikely to have this property.

In some cases, a constant may be subsituted for a register at a loss of effiency – if what was loaded is required to be in a register later on. This is taken care of by a pass in the optimiser which subsitutes constants for registers, with the potential additional benefit of subsituting other, unrelated constants. Additionally, constants take up more space in the instruction encoding than registers, hence we try and reduce the number of constants in the generated code.

In an attempt to exploit features of newer IA32 processors (Pentium, Pentium-II, etc.) `tranx86` attempts to generate new instructions where possible. Currently, this is limited to the '`cmovc`' (conditional move) instruction. For example, the instructions:

```
    cjump CC, Lx
    move <s>, <d>
Lx:
```

reduce into a single "`cmovc CC, <s>, <d>`" instruction.

Some of the more exotic features, such as MMX (Matrix Math Extensions) and SIMD (Single Instruction, Multiple Data) extensions are currently not generated. Using MMX would require more information from the occam compiler about the semantic structure of the program being translated. SIMD instructions are a potential possibility however. These work by performing single-cycle operations on 128-bit registers, which contain packed 16, 32 or 64-bit words, depending on the desired size. Of course, there is a cost associated with the loading and storing of these special registers, but it is likely to be more efficient than the corresponding looping code, especially for multiple operations on the same data elements [17].

2.7 Register allocation

Register allocation is performed in order to allocate the virtual registers generated during the translation into physical registers on the IA32 architecture. The majority of the time, we are only concerned with targetting four common general-purpose registers ('`EAX`', '`EBX`', '`ECX`' and '`EDX`').

The first step is to fix constrained virtual registers into their corresponding physical registers. In some cases, two overlapping virtual registers may be constrained to the same physical register. In these cases, one of the virtual registers is split into two different registers, and the constraint moved to resolve the overlap. In a similar way, any *alternative* real-registers are fixed to physical registers. Alternative real-registers are used in cases where we wish to use a particular physical register, but wish to avoid any interaction with the register allocation. This mainly occurs during run-time error handling (section 2.4), where walking on the program state is harmless.

For each intermediate code block, a graph is built describing the *liveness* of registers in relation to each other. When the number of active registers reaches zero, any generated graph is coloured. In the majority of cases, the graph can be coloured on the first attempt – the *virtual-transputer* stack is only three deep and any graph can be coloured with four colours [18]. The colouring algorithm is currently non-recursive, using a jump to resolve conflicts. A recursive implementation will be used in the future, would enable back-tracking with relative ease[5].

2.8 Code generation

Generating the output code is a relatively simple process. The translated program, after optimisation and register allocation is generated in a form which the GNU assembler under-

[5]The current colouring algorithm is sufficient in that it has handled all the input thrown at it so far. However, a recursive implementation would allow for better searching of a more efficient solution – some IA32 instructions execute faster if the input is in certain registers.

stands. The default output is an actual object file. It does this by invoking the assembler directly, feeding it the assembler code through a pipe.

The code given to the assembler lacks some of the information contained within the intermediate stages however, largely to comply with the assembler syntax. For example, the internal 'add' instruction has three arguments – two inputs and an output. In the generated code, the second input and output are reduced to (and required to be) the same register. This is indeed the case for many IA32 instructions, which use a single argument for both input and output. For different architecture types, e.g. RISC[6], this additional information is likely to be relevant (where there are more registers and the 'add' instruction takes three arguments).

2.9 An example

To illustate the mechanisms involved during translation, the state of a simple occam program is shown as it passes through the translatior, optimiser and register allocator. A simple (sequential) *integrate* process is used, the code for which is:

```
PROC integrate (CHAN OF INT in, out)
  INITIAL INT v IS 0:
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      v := v + x
      out ! v
:
```

After compilation, this is left as an ETC file. Rather than showing the ETC output, figure 8 shows the code after translation and initial optimisation, alongside the ETC input (the PROC entry/exit related code has been removed for clarity).

The optimiser removes the two instructions which only use a single register, those being "move $$x0, %8" and "move 16(%wptr), %12". Being a simple program, no other optimisations are performed. The generated code, after register allocation, is:

```
    movl    $0, 0(%ebp)               -- 'v' := 0
L1:
    lea     16(%ebp), %ebx            -- address of 'x'
    movl    8(%ebp), %eax             -- 'in'
    movl    $0f, -4(%ebp)             -- save return address
    jmp     _Y_in32                   -- jump to input
0:
    movl    0(%ebp), %eax             -- 'v'
    addl    16(%ebp), %eax            -- add 'x'
    into                              -- interrupt if overflow
    movl    %eax, 0(%ebp)             -- store in 'v'
    movl    %ebp, %ebx                -- address of 'v'
    movl    12(%ebp), %eax            -- 'out'
    movl    $0f, -4(%ebp)             -- save return address
    jmp     _Y_out32                  -- jump to output
0:
    jmp     L1                        -- loop
```

[6]Reduced Instruction Set Computing

ETC code	intermediate code	register lifetimes	explanation
`LDC 0`	`move $$x0, %8`	8	load constant 0
`STL 0`	`move $$x0, 0(%wptr)`		store in 'v'
`L1:`	`L1:`		
`LDLP 4`	`lea 16(%wptr), %9`		load pointer to 'x'
`LDL 2`	`move 8(%wptr), %10`	9	load channel 'in'
`IN32`	`move $0f, -4(%wptr)`	10	32–bit input
	`jump _Y_in32`		jump into kernel
	`0::`		saved return–address
`LDL 0`	`move 0(%wptr), %11`		load pointer to 'v'
`LDL 4`	`move 16(%wptr), %12`	12	load pointer to 'x'
`ADD`	`add 16(%wptr), %11, %11`	11	add
	`into`		interrupt if overflow
`STL 0`	`move %11, 0(%wptr)`		store in 'v'
`LDLP 0`	`move %wptr, %13`		load pointer to 'v'
`LDL 3`	`move 12(%wptr), %14`	13	load channel 'out'
`OUT32`	`move $0f, -4(%wptr)`	14	32–bit output
	`jump _Y_out32`		jump into kernel
	`0::`		saved return–address
`J L1`	`jump L1`		jump (loop)

Figure 8: ETC and intermediate code for the '`integrate`' process

3 Inlining for Performance

The default implementation for kernel-involving actions (input, output, ALTernative-wait, etc.) load the necessary parameters into registers or onto the stack, then call an entry-point in the run-time kernel (CCSP [5] in the default case) to perform the action.

Since these operations are essential building-blocks of concurrent applications, efforts have been made to improve their performance through inlining. This basically involves implementing parts, or the whole, of a kernel-call in the generated code.

3.1 Inlined communications and scheduling

The implementations of the input and output instructions follow the transputer algorithms [10], checking the state of the channel word then either performing the communication or blocking the invoking process.

The default translation of the '`IN`' instruction is a call to the relevant kernel entry point, which implements the transputer algorithm. Ultimately, the inputting process will do one of two things when called. If the channel-word is empty, the process inserts itself in the channel word, otherwise it copies the data from the process already in the channel word, before rescheduling it. An inlined version of the '`IN`' kernel-call is shown, where upon entry '`%cptr`' holds the channel address, '`%dest`' holds the destination (data) address and '`%count`' holds the communication size in bytes:

```
    move    $0f, -4(%Wptr)        -- store return-address at Wptr[-1]
    cmp     $0, (%cptr)           -- process waiting ?
    cjump   _NZ_, _Y_fastin       -- yes, jump into the kernel (for copy)
    move    %Wptr, (%cptr)        -- otherwise, place process in channel word
    move    %dest, -12(%Wptr)     -- store destination address at Wptr[-3]
    jump    _Y_fastscheduler      -- call scheduler
0:
```

There are various possible re-arrangements of the above code, but all result in around the same execution time. Two new kernel calls are provided to implement the different actions which can be taken by the input (communicate or block). The '`_Y_fastin`' call copies the data between the two processes, puts the invoking process on the run-queue and returns to the process which was blocked on the channel. The '`_Y_fastscheduler`' call calls the scheduler proper if there are no runnable processes left, or picks the next process off the run-queue and runs it. In the case where another process is picked off the run-queue, no checks are made for timeouts, completed blocking system calls [19] or keyboard input – the process which completes the input will make these checks.

If the *inline-scheduler* option is enabled, then the call to '`_Y_fastscheduler`' above is replaced with the following code:

```
    cmp     $0, Fptr              -- run-queue empty ?
    cjump   _Z_, _X_scheduler     -- yes, call scheduler
    move    Fptr, %Wptr           -- otherwise, load process from run-queue
    move    -8(%Wptr), %tmp       -- load Wptr[next] (next process on queue)
    move    %tmp, Fptr            -- update run-queue
    jump    -4(%Wptr)             -- continue running new process
```

Inlining this part of the scheduler improves performance significantly (around 20%). To improve the scheduling performance even further, it would be possible to remove checks for an empty run-queue all together, which would result in a *page-fault* exception. This could be wired to call the scheduler, which would either sleep waiting for an event, or report a deadlock sitation. Note that there would be no need to recover information about the faulting (occam) process – it would be sitting in a channel word.

3.2 Inlined timer operations

The original implementation of timers in KRoC/Linux was done using the '`gettimeofday`' system-call, which returns the current time in seconds and micro-seconds. This meant that any `TIMER` input in occam caused the Linux kernel to be entered, entailing a relatively large overhead (tens of micro-seconds).

Fortunately, the Pentium family of processors provide a 64-bit *real-time clock*, which is incremented at the processor clock speed and set to zero when the processor is reset. An instruction exists which reads this value into two of the general-purpose registers, which can then be used to calculate the current time in micro-seconds for the occam program. When the system is built on a pre-pentium architecture, using the '`gettimeofday`' system-call is unavoidable, in the majority of cases however, these CPU timers will be available.

In order to use CPU timers, the processor clock speed needs to be known. Rather than attempting to calcuate this each time a program starts (which takes a few seconds for an accurate result), the CPU speed is calculated during the installation of KRoC/Linux and placed in a system-wide file. Rather than dividing the CPU time by the clock speed, a small trick is used to minimise the cost of this calculation (both in the inlining and in the kernel implementation). The value of $(2^{32} \div S_{mhz})$ is calculated initially from the clock speed

(S_{mhz}) and stored in the global variable 'glob_cpufactor', which is used as a multiplier in the actual calculation. The following code shows the implementation of the inlined timer-load (LDTIMER) instruction:

```
rdtsc                            -- read cycle counter into edx:eax
move    %edx, %tmp               -- save high bits
mul     glob_cpufactor           -- multiply eax by factor into edx:eax
move    %tmp, %eax               -- restore high bits into eax
move    %edx, %tmp               -- save high bits of multiplication result
mul     glob_cpufactor           -- multiply eax by factor into edx:eax
add     %tmp, %eax, %eax         -- add high bits of first to low bits
                                 -- of second result
```

This leaves the resultant 32-bit time in micro-seconds in the 'EAX' register. Implied arguments to 'rdtsc' (read time-stamp counter) and 'mul' (unsigned multiply) are not shown. Although the registers shown are physical registers (with the exception of '%tmp'), the intermediate code holds them as virtual registers, constrained to the physical registers shown.

Without the CPU timers, we use a kernel timeout signal and poll a status flag for handling timeouts (timeout guards in ALTs and delayed timer inputs, e.g. in "tim ? AFTER t"). With CPU timers, it is much quicker to poll the CPU timer value (instead of polling the timeout flag set by the timeout signal handler – which is expensive). Dispite the differences in the implementation, the occam world still interacts with both in the same way, i.e. an ordered queue of processes waiting for timeouts called 'Tptr' and some timer-related instructions.

4 Performance

To gauge the performance of tranx86 (and associated optimisations in CCSP) we use a variety of benchmarking programs. The first, 'grantest' tests the ability of the system to execute fine-grained processes. This program shown in figure 9 and was taken from [20]. 'grantest' takes three parameters, defined in an include file ("params.inc"). The results of this benchmark for an array-size ('s') of 2^{16} (64k) are shown in figure 10.

```
#INCLUDE "params.inc"

-- s is the array size
-- g is the process granularity (comp/comms ratio)
-- l is the length of each individual process (no. of comp phases)

PROC main (CHAN OF BYTE in, out, err)
  [s]INT a:
  PAR i = 0 FOR s/g
    CHAN OF INT chan:
    SEQ j = 0 FOR l
      PAR
        SEQ
          SEQ k = 0 FOR g
            a[(g * i) + k] := a[(g * i) + k] + 1
          chan ! i
        INT t:
        chan ? t
:
```

Figure 9: occam benchmark to test KRoC's ability to execute fine-grained programs

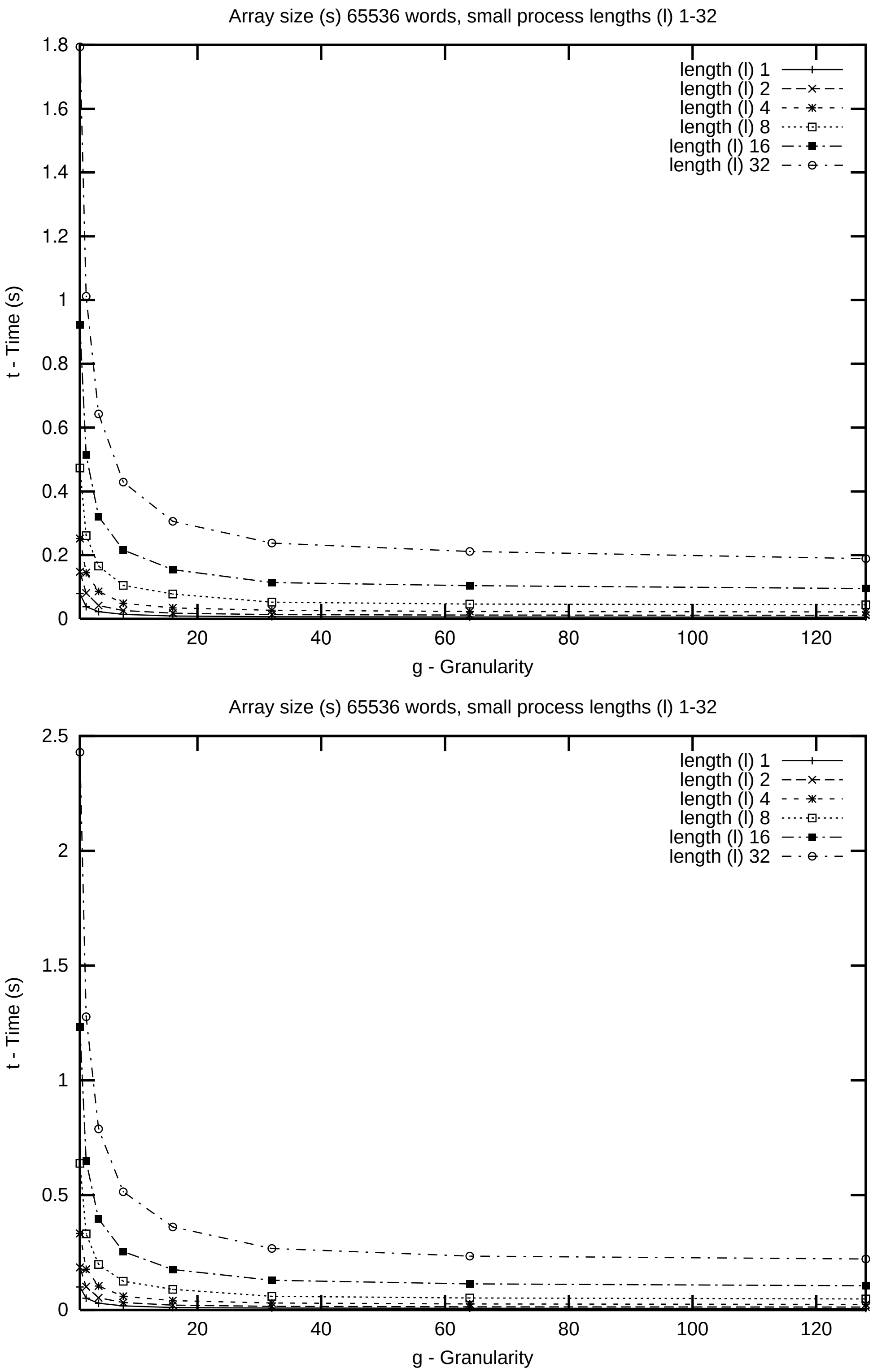

Figure 10: KRoC scheduling overheads for fine-grained processes. The results for the new translator (`tranx86`), with inlining enabled, are shown in the top graph and those for the old translator (`tranpc`) in the bottom graph. The curves take the same shape – but the vertical (time) scales are different.

These graphs (figure 10) show that the KRoC/Linux system using the new translator handles fine-grained processes relatively well when compared with the old translator, especially at very small granularities. Figure 11 shows the percentage increases in execution speed for the new translator over the old one. A reasonable amount of this (especially at lower

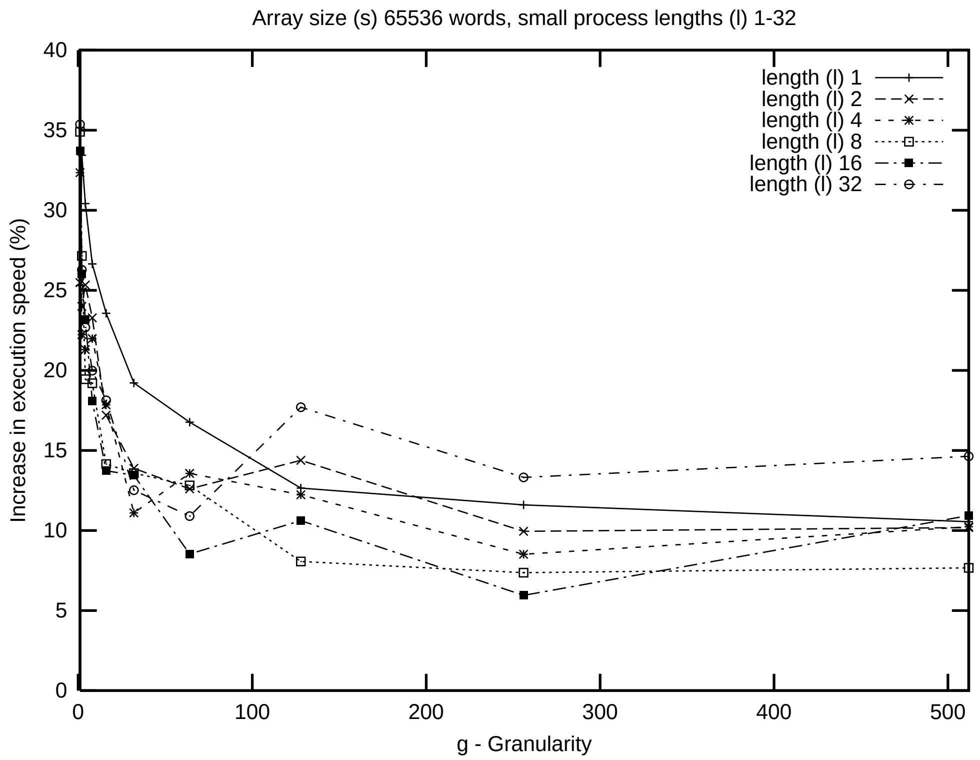

Figure 11: '`grantest`' execution time speedup between the old ('`tranpc`') and new ('`tranx86`') translators. (This is the percentage speedup of the curves in the bottom graph of figure 10 over those in the top.

granularaties) is attributable to an optimisation in the CCSP run-time kernel, which reduced communication from two context switches to one (at the point where the second process engages in the communication) – the previous behaviour was to put both processes on the run queue and reschedule. The new implementation puts the invoking process on the run-queue and immediately schedules the blocked one.

4.1 Commstime

A common benchmark for KRoC systems is '`commstime`' (the code and network illustration for which can be found in [21]), which measures the overhead in channel communication (of one integer) between parallel processes. Table 2 shows the results obtained for the old and new translators. '`tranx86`' is measured twice, once with inlining enabled and once without.

Translator	Channel cost (`SEQ` delta)	Process startup/ shutdown cost
'`tranpc`'	233	49
'`tranx86`'	104	19
'`tranx86`' (inlining)	67	15

Table 2: Commstime benchmark results for new and old translators measured on an 800 MHz Pentium-3. The times are given in nano-seconds.

Again, a large proportion of the reduction in overheads is attributable to the removal of a context-switch on communication. However, when inlining is enabled the cost of a communication is reduced even further (a 35% gain for the inlined version over the non-inlined version).

5 Conclusions and Further Work

tranx86, although still relatively young, provides an efficient translation from ETC generated by the occam compiler into IA32 machine code. With the emergance of IA32 processors as a common computing platform, efficient delivery of CSP (as encompassed by the occam language) is important.

The construction of Networks of Workstations (NoWs) using IA32 based machines opens the door to potential mass distribution of occam programs, to build scalable server farms for example. Whilst the UNIX socket interface is available for building the logical connectivity infrastructure [22], work is in progress to bind a low-level communications architecture into KRoC/Linux directly. This uses the low-level ethernet drivers from MESH [13], which provides direct access to the hardware from user-space for the Intel EEPro-10/100b (fast-ethernet) and Alteon AceNIC (gigabit) cards. Interaction with the occam world is handled using *mobiles* [11, 23], which are allocated inside the low-level communication buffers, as opposed to *mobile-space*.

tranx86 is still largely work in progress. There is much scope for additional optimisations, such as handling loops more efficiently. Currently no effort is made to perform loop-unrolling, cycle reduction [24] or common sub-expression elimination, which would improve performance for many programs. The introduction of support for SIMD and MMX instructions (section 2.6) would probably speed up array data-processing operations considerably. This is currently work-in-progress. An additional approach to optimisation would be to improve the (ETC code) output of the occam compiler itself, as has been done in [25] (but which is unfortunately not available to the community because of commercial licensing costs of the third-party optimiser tool used).

The primary target is currently the i386 family of processors, with a large amount of tranx86 being tied to it – this includes some dependent code in tranx86 itself, used to automatically discover the CPU capabilities (for code-generation) at run-time using the 'cpuid' instruction [2]. Work is in progress however to move more of the IA32 dependent code into the code-generation stage (which is still relatively primative), and provide an additional MIPS target for SGI/Indy hardware running Linux/MIPS [26].

As noted in section 2.3, tranx86 is designed to support different thread schedulers, even though CCSP is currently the only fully supported one. Having this choice makes the job of porting KRoC to different IA32 environments much simpler – the current development release reportedly works correctly on FreeBSD systems. Supporting SMP schedulers also becomes simpler. Critical locks and other synchronisations can be explicitly pre-programmed into the generated code, and tranx86 is designed to make such additions relatively simple.

Acknowledgements

Many thanks to Peter Welch for providing supervision and ideas, and to Jim Moores, Michael Poole and David Wood for providing valuable technical information regarding the existing KRoC systems and translators. Additional thanks to David Wood for pointing out the $(2^{32} \div S_{mhz})$ trick and various octran optimisations. Lastly many thanks to EPSRC and the UKC Computing Laboratory for providing funding, without which this work would not have taken place.

References

[1] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, Concurrent Sys-

tems Engineering, pages 143–166. World occam and Transputer User Group, IOS Press, Netherlands, March 1996. ISBN: 90-5199-261-0.

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

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

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

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

[6] C.A.R. Hoare. Communicating Sequential Processes. In *CACM*, volume 21-8, pages 666–677, August 1978.

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

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

[9] A.Ripke T.Sheen, A.R.Allen and S.Woo. oc-X: an optimising multiprocessor occam system for the PowerPC. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 167–186, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press.

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

[11] F.R.M. Barnes and P.H. Welch. Mobile Data Types for Communicating Processes. In *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, June 2001.

[12] Richard M. Stallman. *Using and Porting GNU CC: Version 2.8*. Free Software Foundation, March 1998.

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

[14] Richard Beton. libcsp – A Binding Mechanism for CSP Communication and Synchronisation in Multithreaded C Programs. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 239–250, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press.

[15] D.C. Wood and F.R.M. Barnes. Post-Mortem Debugging in KRoC. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 179–192, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press.

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

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

[18] K. Appel and W. Haken. Every Planar Map is Four Colorable. In *Bulletin of the American Mathematical Society*, volume 82, pages 711–712, 1976.

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

[20] Kevin Vella. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations*. PhD thesis, The University of Kent at Canterbury, Canterbury, Kent. CT2 7NF, December 1998.

[21] R.M.A. Peel. Parallel Programming for Hardware/Software Co-Design, February 2001. Available at:
`http://www.ee.surrey.ac.uk/Personal/R.Peel/bcs-220201-4.pdf`.

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

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

[24] Susan L. Graham David F. Bacon and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. Technical report, Computer Science Division, University of California, Berkeley, California 94720, 1993. Technical Report No. UCB/CSD-93-781.

[25] Spiridon Kalogeropoulos. Developing an Optimising Compiler for occam. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 157–166, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press.

[26] Ralf Bchle. Linux/MIPS HOWTO, April 2001. Available at:
`http://oss.sgi.com/mips/mips-howto.html`.