Performance Improvement with the INMOS Dx305 occam 2 Toolset

INMOS is a member of the SGS–THOMSON Microelectronics Group

INMOS document number: 72 TDS 379 00

Contents

Preface

Host versions

The documentation set which accompanies the occam 2 toolset is designed to cover all host versions of the toolset:

- IMS D7305 – IBM PC compatible running MS–DOS
- IMS D4305 – Sun 4 systems running SunOS.
- IMS D6305 – VAX systems running VMS.

About this document

'Performance Improvement with the DX305 occam 2 Toolset'

This document provides advice about how to maximize the performance of the toolset. It brings together information provided in other toolset documents particularly from the *Language and Libraries Reference Manual.*

The document describes the layout of code and data in memory for programs developed with the DX305 occam 2 Toolset. It then goes on to describe methods of improving code in order to:

- minimize the running time of the program;
- reduce the size of the program; either code or data or both.

Note: details of how to manipulate the software virtual through-routing mechanism are given in the *User Guide.*

About the toolset documentation set

The documentation set comprises the following volumes:

- *72 TDS 366 01 occam 2 Toolset User Guide*

  Describes the use of the toolset in developing programs for running on the transputer. The manual is divided into two sections; '*Basics*' which describes each of the main stages of the development process and includes a '*Getting started*' tutorial. The '*Advanced Techniques*' section is aimed at more experienced users. The appendices contain a glossary of terms and a bibliography. Several of the chapters are generic to other INMOS toolsets.

- *72 TDS 367 01 occam 2 Toolset Reference Manual*

  Provides reference material for each tool in the toolset including command line options, syntax and error messages. Many of the tools in the toolset are generic to other INMOS toolset products i.e. the ANSI C and FORTRAN toolsets and the documentation reflects this. Examples are given in C. The appendices provide details of toolset conventions, transputer types, the assembler, server protocol, ITERM files and bootstrap loaders.

- *72 TDS 368 01 occam 2 Toolset Language and Libraries Reference Manual*

  Provides a language reference for the toolset and implementation data. A list of the library functions provided is followed by detailed information about each function. Details of extensions to the language are given in an appendix.

- *72 TDS 379 00 Performance Improvement with the DX305 occam 2 Toolset* (this document)

- *72 TDS 377 00 occam 2 Toolset Handbook*

  A separately bound reference manual which lists the command line options for each tool and the library functions. It is provided for quick reference and summarizes information provided in more detail in the *Tools Reference Manual* and the *Language and Libraries Reference Manual.*

- *72 TDS 378 00 occam 2 Toolset Master Index*

  A separately bound master index which covers the *User Guide, Toolset Reference Manual, Language and Libraries Reference Manual* and the *Performance Improvement* document.

Other documents

Other documents provided with the toolset product include:

- Delivery manual giving installation data, this document is host specific.
- Release notes, common to all host versions of the toolset.
- '*occam 2 Reference Manual*' published by Prentice Hall.
- '*A Tutorial Introduction to occam Programming*' published by BSP Professional Books.

FORTRAN toolset

At the time of writing the FORTRAN toolset product referred to in this document set is still under development and specific details relating to it are subject to change.

Documentation conventions

The following typographical conventions are used in this manual:

Bold type	Used to emphasize new or special terminology.
`Teletype`	Used to distinguish command line examples, code fragments, and program listings from normal text.
Italic type	In command syntax definitions, used to stand for an argument of a particular type. Used within text for emphasis and for book titles.
Braces { }	Used to denote optional items in command syntax.
Brackets [ ]	Used in command syntax to denote optional items on the command line.
Ellipsis . . .	In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.
|	In command syntax, separates two mutually exclusive alternatives.

1 Introduction

This document describes ways in which the performance of occam programs which have been built with the INMOS D4305, D6305, and D7305 occam toolsets can be improved. It assumes that the reader is familiar with these toolsets, and has the user documentation available. It supersedes INMOS Technical note 17 (“*Performance Maximisation*”), which was also supplied as chapter 18 of “*The Transputer Applications notebook - Systems and Performance*”. Some of the information in this document is taken directly from that technical note.

Many of the techniques described here are equally applicable to early occam toolsets, and indeed to many other languages and computer systems. Similarly, many of the transputer-specific optimizations are relevant when programming transputers in other languages.

Also discussed are the specific features of the transputer which are amenable to optimizations, and how to use the toolsets to take advantage of them. Transputers covered include the T212, T222, T225, T400, T414, T425, T800, T801, T805.

1.1 Transputer architecture

This document will not attempt to describe the transputer architecture. However, the particular points to note about the transputer, when considering performance improvement, are as follows:

- On-chip RAM.

  Each transputer has a part of its address space implemented as on-chip RAM, which means that it can be accessed very quickly. There is a noticeable penalty in accessing external RAM. Much of this document describes methods to ensure that best use is made of this on-chip RAM. Current transputer variants have either 2K or 4K bytes of on-chip RAM.

- Instruction prefixing.

  Transputers use a variable length instruction encoding, which is built up out of lots of single byte instructions. It is useful to minimize the size of these instructions, both to minimize the code space required, and to minimize the time taken to fetch the instructions from memory.

In practice, the only instructions whose length can be easily controlled are those which access local variables; hence it is the layout of local variables which is important.

2 Trade-offs and issues

2.1 Space versus time

Most optimizations which are performed are intended to minimize the running time of a program. This is known as optimizing for time. In certain circumstances it is required to minimize the size of a program; either code size, data size, or both. This is known as optimizing for space. Often a particular optimization will produce an improvement in both space and time. In general, and in this document, most optimizations are aimed to optimize for time.

2.2 On–chip RAM

The on-chip RAM is based at the bottom of a transputer's memory address space. This implies that it is important to utilize this space properly. In some cases, the whole program can fit inside this RAM. In many other cases, however, a decision must be made as to the best use of this RAM.

As a general rule of thumb, a program's *stack* (or *workspace*) should be placed onto the on-chip RAM if possible. This is because the transputer's instruction encoding makes data accesses more frequent than instruction accesses, so less penalty will be incurred if the data resides in fast memory. If there is space, then it is useful to put inner loops and other frequently used code subroutines into on-chip RAM too.

The INMOS occam toolsets attempt to place workspace onto on-chip RAM by default.

Some TRAMs (Transputer Modules) are constructed in such a way that the 'next' memory in the address space is the next fastest, followed by slower RAM at a higher address. Thus the bottom 4K might be on-chip RAM, the next 32K might be 3-cycle external SRAM, followed by 2M of 4-cycle external DRAM. The way these are treated is exactly analogous to the simpler case; by simply attempting to move the most important data and code areas to the bottom of memory.

2.3 Basic code generation techniques

The compiler supplied with the current INMOS occam toolsets generates good code for expressions, but does not attempt to optimize code across statement boundaries; future compilers will. It may be useful to bear this in mind to try to improve performance; for example, by the introduction of abbreviations for commonly used expressions.

The occam compiler allocates memory statically; that is, given any program the compiler can determine exactly how much memory is required. This enables the loader to specify exactly how much workspace is required and to attempt to place it on-chip.

2.4 Processor classes and types

The occam 2 compiler can create code which can execute on many different types of transputer; these are known as transputer classes. This facility can be useful to build libraries which can be used for any transputer type. However, compiling for a particular transputer type will make most use of that transputer's particular instruction set, and therefore will make a program execute faster.

It is worth noting that you can create a library which contains, for example, both TA and T425 code. The compiler and linker will automatically select the most specific modules which exist in that library, depending on the command line options supplied to the compiler or linker. Similarly if both interactive debugging and non-interactive debugging modules exist in a library, the most specific one (non-interactive) will be chosen.

The rule to use is: Always compile and link for the specific transputer type to get the best performance.

2.5 Interactive debugging

The INMOS toolsets allow an interactive debugger to execute a program under user control, to allow breakpointing, etc. This involves a certain amount of overhead, both of space and time, because the compiler inserts run-time library calls for certain operations; these require extra code, and extra stack space. Once a program has been developed, this facility can be disabled to improve program performance.

Disabling interactive debugging has two effects; firstly, it will improve the speed of code which performs communication. This is because the interactive debugger requires (nearly) all communication to be performed via library routines. The second effect is to reduce the memory size of the program; this reduces both workspace size (because the library routines would use stack space) and code size (because the library routines are not linked in). Also, when interactive debugging is enabled, a generic start-up routine must be inserted. Configuring a program with interactive debugging disabled, or collecting a single processor program with interactive debugging disabled, means that simpler start-up code can be used.

The rule to use is: Always compile, link and configure with interactive debugging disabled to get the best performance.

2.6 Virtual routing

This occam configurer provides the ability to use *virtual routing*. This means that messages from one transputer to another can be automatically through-routed via intervening transputers without requiring to be explicitly programmed. Also, many occam channels may be multiplexed down a single transputer link.

This is performed by library routines which are inserted by the compiler. These are the same routines as are used by the interactive debugging system described in section 2.5. Therefore, to use the virtual routing facilities you must not disable interactive debugging.

When virtual routing is used, extra processes are placed onto each processor by the configurer as required. These will use up memory. Therefore if memory space is at a premium, it will be better not to use the virtual routing facilities. The configurer will automatically determine when the multiplexing and through-routing are not required, and will omit the processes. A command line switch '`NV`' on the configurer can be used to forcibly prevent the creation of these processes; in this case an error is issued if they would have been required.

2.7 Error modes

occam programs can be compiled in three different error modes; HALT, STOP, and UNIVERSAL. In HALT mode, as soon as any erroneous process is executed, the whole processor halts. This is implemented by making use of the transputer's global **HaltOnError** flag. In STOP mode, only that single process is halted. This requires the **HaltOnError** flag to be clear. UNIVERSAL mode is provided to act like either HALT or STOP, depending on the state of the **HaltOnError** flag.

Because of the behavior of particular transputer instructions, HALT error mode is the simplest and fastest to implement. STOP mode requires extra instructions to be inserted to detect and act on errors. UNIVERSAL mode requires slightly more instructions again.

UNIVERSAL mode does *not* switch off all run-time error checking. The `U` command line switch can be used to remove error checking code (see section 4.1). Thus any error mode, in combination with the `U` option, provides the occam UNDEFINED error mode.

The compiler libraries are most efficient in the HALT error mode, so benchmark programs, and programs *which are known to be correct*, should be compiled in HALT error mode, and with the `U` option.

The rule to use is: Always compile, link and configure in the HALT error mode, and compile and configure with the `U` option, to get the best performance.

2.8 Vector space

The occam compiler uses a technique known as *separate vector space* to try to minimize a program's stack requirement. Arrays are placed into a separate area of memory known as the vector space. This is organized as another stack, in another area of memory. The idea is that the normal stack (workspace), containing local variables and the procedure call stack, will then be as small as possible, and will fit into on-chip RAM. In many cases the time required to access arrays will be less critical than the time to access local variables, so this provides a useful optimization.

The use of separate vectorspace requires that an extra (hidden) parameter is passed to each subroutine. In some circumstances this extra cost exceeds the benefit, so it might be useful to disable the separate vectorspace (using the compiler `V` option). Similarly, if the combined workspace and vectorspace would together fit into on-chip RAM anyway, it will be most efficient to disable the separate vectorspace.

It may be the case that access to most arrays is not critical, but that access to a particular array is extremely time critical. This single array can be retained in workspace (and hence more likely to be placed onto on-chip RAM) by a compiler *allocation*:

```
PLACE name IN WORKSPACE.
```

Alternatively, a program may consist of many small arrays which would benefit from being placed in workspace, plus a few large arrays which would not. In this case, separate vector space can be disabled by default. The large arrays can then be explicitly placed into vectorspace by another *allocation*:

```
PLACE name IN VECSPACE.
```

Finally, the configurer provides attributes which allow the whole vectorspace to be placed at the bottom of memory, etc. See section 4.5.

2.9 Alias checking

occam has strict rules about aliasing of variables (that is, making two different variables point to the same data). These rules are there for a purpose; they enable the compiler to make better deductions about the behavior of a program, and therefore to generate better code. They also provide a simple model of what a section of program means; its behavior is not affected by the context in which it is executed.

The INMOS occam toolset permits a programmer to break these rules, but at the programmer's own risk. In general, it will be better not to. There are two levels of control over alias checking:

- On a whole program at a time.

  Alias checking may be disabled for a whole program, by using the '`A`' command line switch, or `#OPTION`. The compiler will assume that all variables may alias each other.

- For individual variables.

  `#PRAGMA PERMITALIASES` may be used to indicate individual variables which may be aliased. All other variables are assumed to abide by the occam rules.

By default, the compiler will ensure that no aliases are permitted. In some cases, this can require code being generated to check at run-time. The '`wo`' option will cause a warning message to be generated whenever a run-time check is inserted. Where alias checks are disabled, these checks will not be generated. However, the compiler will have to make worse assumptions about the behavior of the program, and may generate slower code.

2.10 Usage checking

occam has strict rules about the usage of variables and channels in parallel processes. These rules are there for a purpose; they enable the compiler to make better deductions about the behavior of a program, and therefore to generate better code. They also provide a simple model of what a section of program means; its behavior is not affected by the context in which it is executed.

The INMOS occam toolset permits a programmer to break these rules, but at the programmers own risk. In general, it will be better not to. There are two levels of control over usage checking:

- On a whole program at a time.

  Usage checking may be disabled for a whole program, by using the '`N`' command line switch, or `#OPTION`. The compiler will assume that all variables may be accessed in parallel, synchronized by communications down channels.

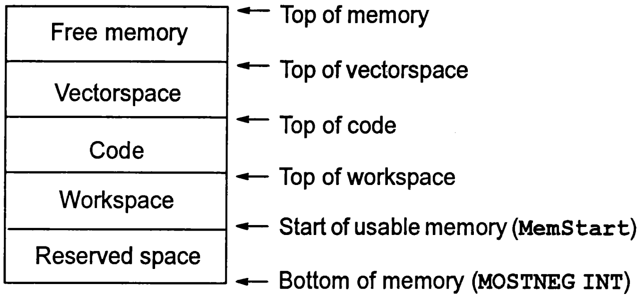
- For individual variables.

  `#PRAGMA SHARED` may be used to indicate individual variables which are used in parallel processes. All other variables are assumed to abide by the occam rules.

By default, the compiler will ensure that no variables and channels are permitted to break the usage rules. Where the checks are disabled, the compiler will have to make worse assumptions about the behavior of the program, and may generate slower code.

2.11 Memory layout

By default, the toolset arranges memory as follows:



When the collector's `S` option is used for single processor programs, the collector allocates another buffer below the workspace. See section 4.6.

The configurer's `order.code`, `order.ws`, and `order.vs` attributes may be used to override this default ordering. By default; the 'reserved space' is simply that memory up to `MemStart`. However the configurer's `reserved` attribute may be used to override this. `location` attributes may also be used to override this memory layout. See section 4.5.

2.12 When there isn't enough on–chip RAM

When a program's workspace is substantially larger than the on-chip RAM, it may well be true that most of the time the stack is working off-chip. In this case, it might be more useful to move the code on-chip, particularly time-critical code sections, and leave all the workspace off-chip. See section 4.3 and section 4.5.

3 Obtaining information

Various tools provide information which can be useful when improving performance.

- Compiler information

  The compiler's `I` command line switch displays information about the workspace and vectorspace requirements of each externally visible procedure or function. It also displays the number of bytes of code in the module.

  The compiler's `P` command line switch can be used to supply the name of a text file which the compiler produces known as the *map file*. This lists the layout of stack memory for each routine in the program, and the layout of the code for each routine. This file can be processed by the `imap` tool to produce a map of the entire program's memory.

- Linker information

  The linker's `MO` command line switch can be used to produce a text file which indicates how the code has been linked together. This file can be processed by the `imap` tool to produce a map of the entire program's memory.

- Collector information

  The collector's `P` command line switch can be used to produce a text file which indicates the memory layout of each processor in the network. It also indicates the processor connectivity. This file can be processed by the `imap` tool to produce a map of the entire program's memory.

- The debugger

  The debugger can be used to provide information about the memory map of a program. The Monitor Page's `M` (Memory map) command will display the memory map of each processor in the network. The `L` (Links) command will indicate the link connectivity.

- The lister tool

  The `ilist` program can examine any data file which is created by the INMOS toolsets, and display a decoded form of its contents. This may be useful if extra information is required which is not available by the previous methods.

- The mapper tool

  The `imap` program takes the map files created by the compiler, linker, and collector, and combines the information into a single text file which lists the whole program's memory layout for each processor.

4 Command line switches

There are many different switches and commands which can be used with the compiler, linker, collector, and configurer, in order to modify program execution speed.

4.1 Compiler command line switches

The following compiler command line switches can affect program performance. Note that many options can be specified in the source code by inserting a `#OPTION` statement as the first line of the program.

A typical benchmark program would be compiled with options `H`, `NA`, `U`, and `Y`, and maybe `A` and `V` too.

`H` HALT error mode. (This is the default)

See the discussion on error modes in section 2.7.

`S` STOP error mode.

See the discussion on error modes in section 2.7.

`X` UNIVERSAL error mode.

See the discussion on error modes in section 2.7.

`K` Disable range checking.

This removes any code whose sole purpose is to check for array bounds violations. Any such check which can be performed at compile time (e.g. because the index is a constant) will still be performed. Note that the `K` option will also disable run-time alias checks.

`U` Disable run-time error checking.

This removes any code whose sole purpose is to detect errors (except `ASSERT` - see below). For example, code to check that the number of replications of a replicator is not negative will be omitted. It does *not* mean that errors are 'allowed'. Some errors may still be detected because the fastest code includes error checking. (E.g. `adc` is used to add a constant number; this instruction performs overflow checking). Note that this option is stronger than the `K` option, in that it does everything that the `K` option does, and more, so it is not necessary to specify both on the command line. Also see the discussion on error modes in section 2.7.

`NA` Disable run-time `ASSERT` checks.

The `ASSERT` predefine can be used to provide security checks. If a check can be performed at compile-time, it will be. Otherwise code is inserted to perform the check at run-time. This option disables the run-time checks.

`N` Disable Usage checking.

See the discussion on usage checking in section 2.10.

`A` Disable Alias checking.

See the discussion on alias checking in section 2.9. Note that the `K` option will also disable run-time alias checks.

`V` Disable vectorspace.

See the discussion on separate vectorspace in section 2.8.

`Y` Disable interactive debugging.

See the discussion on interactive debugging in section 2.5. Also see the discussion on virtual routing in section 2.6.

`P` Produce map file.

This option can be used to specify the name of a text file (the ‘map’ file) which is created by the compiler to provide information about memory layout. This is used by the `imap` tool.

4.2 Linker command line switches

Use the linker's `H`, `S`, or `X` flags to link in the correct error mode. If interactive debugging and virtual routing are not required, disable interactive debugging by using the `Y` flag.

The `MO` option can be used to specify the name of a text file (the 'module' file) which is created by the linker to provide information about the linkage. This is used by the `imap` tool.

4.3 Linker directives

The linker allows a programmer to control the relative ordering of different modules in the linked object file. The output file will still be a single consecutive chunk of code, but the relative order of subroutines can be controlled. Primarily this is done by rearranging the order in which the files are listed on the command line. The linker does, however, provide finer control than this if required.

The linker inserts all separately compiled units into the output code file in the same order as they are encountered on the command line. The first module will be loaded at a lower address, that is, nearer `MOSTNEG INT`. It then adds library modules as necessary. It chooses the entry name of the first separately compiled module to be the entry point of the whole file; normally the top level module is listed first. If you re-order the files on the command line, you must provide a `#mainentry` command to the linker (or use the `ME` command line option) to tell the linker the name of the main entry point of the program.

`#section` directives in the linker's input file provide finer control. By default, the occam compiler places all code in any compilation module into a 'code section' named "`text%base`". This may be overridden by use of the compiler's `#PRAGMA LINKAGE`. If this pragma is specified, the code section is named "`pri%text%base`". If the pragma is followed by a string (in double quotes), that name is used for the code section.

The linker links all code modules in any particular named section in arbitrary order, and then concatenates the sections. However, by naming different sections, a programmer can control the overall order. Normally, the linker places the section named "`pri%text%base`" at the beginning of the code, (i.e. nearer `MOSTNEG INT`), followed by "`text%base`", followed by any other code sections. If the programmer supplies *any* `#section` directives in the linker's input file, the default is ignored. Instead, the linker places the first named section first, followed by the next named section, etc. Any sections which were not explicitly named are placed at the end. (Note that the `#section` directive should be followed by the section name *without* enclosing quotes). The module file created by the linker's `MO` option can be examined to confirm the relative placement of sections.

Note that floating point support libraries used on `T4` series transputers are automatically placed into section "`pri%text%base`", so that they are more likely to be placed onto on-chip RAM.

4.4 Configurer command line switches

Interactive debugging should be disabled using the '`Y`' option, for optimum performance. This also means that a smaller start-up routine can be used, which enables more of a user's program to fit into on-chip RAM.

Similarly, virtual routing should be disabled by using the '`NV`' option if appropriate.

4.5 Configuration language attributes for optimizing memory

As described in the discussion on memory layout in section 2.11, by default workspace is placed at the lowest address, followed by code, followed by vectorspace. The configurer provides attributes which can be set to override this default.

4.5.1 Ordering attributes

The `order.code` attribute of a `PROCESSOR` can be set to an integer value, in the `MAPPING` section of the configuration source. The default value is 0. If this attribute is set to a value less than 0, the code will be placed at a lower address than the workspace. Similarly, the `order.vs` attribute can be set to a negative value to indicate that it should be placed at a lower address than the workspace. The relative values of `order.code`, `order.ws`, and `order.vs` indicate which should be placed at a lower address.

The collector's map file (produced by the `p` command line option) can be inspected to see the effect of these switches.

Note that this facility must be explicitly enabled by the configurer's `RE` option, because the `idebug` debugger cannot be used if the memory layout has been altered.

4.5.2 Location attributes

Attributes `location.code`, `location.ws`, and `location.vs` can be used to explicitly specify where the code, workspace, or vectorspace of a program should be placed. They are set to a machine address. This address must not be within the address range used by the configurer for its own use; namely that area above `reserved` `BYTE`s from MOSTNEG INT, and below `memsize` `BYTE`s from MOSTNEG INT.

4.5.3 Reserved attribute

The `reserved` attribute can be used to tell the configurer not to use the memory immediately above MOSTNEG INT; i.e. the lowest addresses in memory. It should be set to the number of `BYTE`s above MOSTNEG INT which are to be reserved. The configurer will then leave this area free for use by the programmer; either by using the `location` attributes, or by `PLACE`ing data there directly.

4.6 Collector command line switches

`Y` Disable interactive debugging.

This also allows the collector to insert a more specialized loader which takes up less memory. (This option is not required when collecting a program which has already been configured with the `Y` option).

`M` Specify memory size.

When collecting a program for a single transputer, this option creates a bootable which does not examine the environment variable `IBOARDSIZE` at run-time. Instead, it uses the value supplied to the collector. This will reduce the memory requirements of the start-up code.

`P` Specify memory map file.

This option creates a map file describing the memory map of each processor, and their connectivity. This is used by the `imap` tool.

`S` Specify stack size.

This option can be used for single processor programs to gain access to a reserved block of memory at the bottom of the address space. (In the C toolset it is used to place the C program's stack onto the on-chip RAM, hence the name of the option).

A single processor program normally requires the following formal parameter list:

```
#INCLUDE "hostio.inc"
PROC myprog (CHAN OF SP fs, ts, []INT freespace)
```

This can be modified as follows:

```
#INCLUDE "hostio.inc"
PROC myprog (CHAN OF SP fs, ts, []INT freespace, []INT buffer)
```

The collector will allocate a buffer at the bottom of memory, and pass it as the extra parameter. If the collector's `S` option is not specified, this array will be of length zero. The buffer can be used to hold an array of data which is required to be accessed quickly. See the discussion about memory layout in section 2.11.

5 Compiler optimizations

The compiler already performs some minor optimizations:

- Constant folding

  Expressions such as `27 + 33` are 'folded' into `60`.

- Unused variable elimination

  Any variables which are never used are not allocated stack space.

- Dead code elimination

  Branches of `IF`s which can be determined at compile-time to always be FALSE are ignored.

- Constant tables

  The transputer instruction set creates large constants by repeated use of a `pfix` instruction. This can sometimes mean that loading large constant values can be slow. However, the compiler recognizes such constants, and 'caches' them into a constant table, which is accessed quickly via a small offset from a local pointer.

- Workspace allocation

  The compiler estimates the usage frequency of each variable which is used in a procedure or function, and allocates the variables in memory so that the most used variables are stored at small offsets in workspace, so as to minimize the overall execution cost.

- Merging of constant arrays

  Constant arrays and strings which appear more than once in a single compilation unit are merged into a single array, and thus only appear once in the code.

- Elimination of temporaries

  The compiler 'knows' that different variables always refer to different data items, and can reduce the number of temporaries required in multiple assignment, etc.

6 Source code optimizations

The reader should be aware that many of the following source code optimizations are implementation dependent, and may actually result in a performance degradation in a different implementation of the occam language.

6.1 Compiler workspace layout

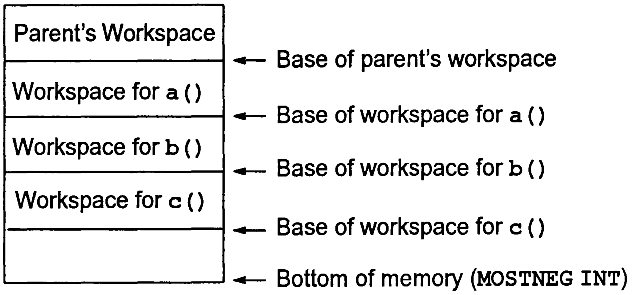
The current compiler allocates workspace as a falling stack. Hence the workspace for a nested procedure or function will be allocated at a lower address than that of the enclosing subroutine.

Workspace for parallel processes are allocated below the workspace of the parent. The first member of the `PAR` list (or the lowest replicator value of a replicated `PAR`) is allocated workspace immediately below the parent, the next immediately below that, etc. Thus the *last* process will have the lowest workspace address, and hence is most likely to be placed on-chip.

Suppose we have three procedures `a()`, `b()` and `c()`. Then

```
SEQ
  ...  body of parent
  PAR
    a ()
    b ()
    c ()
```

is allocated as follows:


Parent's Workspace
Base of parent's workspace
Workspace for a()
Base of workspace for a()
Workspace for b()
Base of workspace for b()
Workspace for c()
Base of workspace for c()
Bottom of memory (MOSTNEG INT)

This arrangement also holds for `PRI PAR`. Hence if it is important to get the workspace of a high priority process on-chip, where the low priority process has a large workspace, the following can be used:

```
PAR
  ...  low priority process
  PRI PAR
    ...  high priority process
    SKIP
```

6.2 Compiler code layout

The compiler writes code for a single compilation into a single object file. PROC and FUNCTION bodies are written in the *reverse* order of the *end* of their declaration; thus if you read the source backwards, the routines are inserted into the object file in the order in which you find their terminating colons (:). This means that all calls are *forwards*, and that calls to a routine do not have to jump over the body of that routine; these considerations help make the call instruction smaller.

However, it tends to mean that nested subroutines are placed at higher addresses, which can push them out of on-chip RAM. It may be useful to make critical inner subroutines into separately compiled units, and use the linker to place that at a lower address (see section 4.3).

6.3 Abbreviations

Abbreviations are a powerful feature of the occam language. They can be used to bring non-local variables down into local scope, thus removing the need to chain through the procedure call stack, and speeding up access. They can also speed up execution by removing range check instructions. Where appropriate, VAL abbreviations should be used; for scalar values this creates a local copy of a variable rather than a pointer to it.

6.3.1 Abbreviations — They should not be too trivial

When performance is the main aim, abbreviations should not be used if the value is only used once or twice. Similarly, if an expression is simple, it may be faster to re-evaluate the expression, rather than to read another value from memory, especially if the workspace does not fit in on-chip RAM.

6.3.2 Abbreviations — Removing range–checking code

By abbreviating sub-vectors of larger vectors and using constants to index into the sub-vector, the compiler will generate range-checking code for the abbreviation, but will not need to generate range-checking code for accesses to the sub-vector.

As an example of an abbreviation removing range check instructions, here are two versions of the same procedure. Part of the ray-tracer, this procedure is initializing fields in a new node to be added into a tree. The identifier nodePtr points to the start of the node. The second version uses abbreviations, generates no range checking code (apart from initial generation of the abbreviation) generates shorter code sequences for each assignment, and executes more quickly.

```
PROC initNode ( VAL INT nodePtr )    -- version 1
  SEQ
    tree [ nodePtr + n.reflect] := nil
    tree [ nodePtr + n.refract] := nil
    tree [ nodePtr +    n.next] := nil
    tree [ nodePtr +  n.object] := nil
:
PROC initNode ( VAL INT nodePtr )    -- version 2
  node IS [ tree FROM nodePtr FOR nodeSize ] :
  SEQ
    node [ n.reflect] := nil
    node [ n.refract] := nil
    node [    n.next] := nil
    node [  n.object] := nil
:
```

Even if range-checking were switched off, the second version will execute more quickly. Without range check instructions, the statement will generate the following transputer instructions:

```
ldc   nil        -- get data to save
ldl   nodePtr    -- get pointer to base of node
ldl   static     -- get static chain
ldnlp tree       -- generate pointer to tree ( in outer scope)
wsub             -- generate pointer to tree [ nodeptr]
stnl  n.refract  -- and store to tree [ nodePtr + n.refract]
```

whereas the second version will generate the following, appreciably shorter and faster fragment of code:

```
ldc   nil        -- get data to save
ldl   node       -- load abbreviation
stnl  n.refract  -- and store
```

Of course there is an initial overhead to generate the abbreviation, but this is rapidly swamped by the subsequent savings.

6.3.3 Abbreviations — Loop unrolling

Using abbreviations in conjunction with loop unrolling by hand can speed up execution considerably. Take the following piece of occam , a simple vector addition:

```
SEQ i = 0 FOR 20000
  a[i] := b[i] + c[i]
```

The transputer loops in about a microsecond, but adds in about 50 nanoseconds. Therefore to increase performance we must increase the number of adds per loop:

```
VAL bigLoops IS 2000 / 16 :
VAL leftOver IS 2000 - (bigLoops TIMES 16) :
SEQ
  SEQ i = 0 FOR bigLoops
    VAL base IS i TIMES 16 :
    aSlice     IS [ a FROM base FOR 16 ] :
    VAL bSlice IS [ b FROM base FOR 16 ] :
    VAL cSlice IS [ c FROM base FOR 16 ] :
    SEQ
      aSlice  [0] := bSlice [0] + cSlice [0]
      aSlice  [1] := bSlice [1] + cSlice [1]
      aSlice  [2] := bSlice [2] + cSlice [2]
                 ............
      aSlice [14] := bSlice[14] + cSlice[14]
      aSlice [15] := bSlice[15] + cSlice[15]
  SEQ i = 2000 - leftOver FOR leftOver
    a[i] := b[i] + c[i]
```

Obviously, loops can be opened out in any language, on any processor, and performance will tend be improved at the expense of increased code size. However, opening loops out in slices of 16 has a knock-on effect on the transputer, as optimal code **with no prefix instructions** is generated for each addition statement. Compare the code generated for the two statements:

```
                                        ldl  i
                                        ldl  b
                                        wsub
                                        ldnl 0
                                        ldl  i
a[i] := b[i] + c[i]                     ldl  c
                                        wsub
                                        ldnl 0
                                        add
                                        ldl  a
                                        ldl  i
                                        wsub
                                        stnl 0


                                        ldl  bSlice
                                        ldnl 15
                                        ldl  cSlice
aSlice[15] := bSlice[15] + cSlice[15]   ldnl 15
                                        add
                                        ldl  aSlice
                                        stnl 15
```

The second piece of code is just over half the size of the first and the number of loop end (`lend`) instructions executed is reduced by a factor of 16.

6.4 Vectorspace

Use the *allocations:*

`PLACE name IN WORKSPACE` and `PLACE name IN VECSPACE` as described in the discussion on separate vector space (section 2.8).

Suppose we wish to clear a large block of memory, such as a clear screen operation. It may be worthwhile using an array which is placed in on-chip RAM as the source of a block move:

```
PROC ClearScreen(VAL BYTE pattern)
  -- the screen is declared as [512][512]BYTE screen :
  [SIZE screen[0]]BYTE fastvec :  -- this is in on-chip RAM
  PLACE fastvec IN WORKSPACE :
  SEQ
    initBYTEvec(fastvec, pattern) -- fast BYTE initializer
    SEQ y = 0 FOR SIZE screen
      screen[y] := fastvec        -- use a block move
:
```

This fires off 512 block move instructions, each of 512 bytes. Since the block move is reading from on-chip memory, and writing to off-chip memory, it will proceed more quickly than

```
PROC ClearScreen(VAL BYTE pattern)
  -- the screen is declared as [512][512]BYTE screen :
  [(SIZE screen) * (SIZE screen[0])]BYTE bytescreen :
  initBYTEvec(bytescreen, pattern) -- fast BYTE initializer
:
```

where all data accesses are to off-chip memory. The time saved during the block moves outweighs the cost of setting up the parameters to the block moves, and of the initial `initBYTEvec`. See section 6.7 for more about block moves and for the source of `initBYTEvec`.

6.5 Beware the PLACE statement

A common mistake in trying to make occam go faster is to physically place data on-chip, using a `PLACE AT` statement. This does the right thing - the compiler will physically place the variable on-chip, but the variable will be outside local workspace. Therefore to access the variable, its physical address must be generated, and an indirection performed to load the contents of the address. For example, declaring a variable at word address 30 above MOSTNEG INT, and setting its value to 3 :–

```
INT a :
PLACE a AT 30 : -- 30th word above MOSTNEG INT
a := 3

ldc 3
mint
ldnlp 30
stnl 0
```

This code sequence takes 7 cycles (350 ns on a T800-20). Were `a` a local variable, the code sequence would take only 3 cycles (150 ns) if the workspace were on-chip, and would be:

```
ldc 3
stl a
```

In any case it is dangerous to place variables directly into on-chip RAM, because unless the on-chip RAM has been reserved in some other way, the explicit allocation to on-chip RAM will clash with some other code or data which is already there.

The key to making variable accesses go faster is to **keep the workspace on-chip**. Then if it is necessary for a vector to be on-chip, it can be declared in local scope.

6.6 Abbreviating PLACED objects

In some circumstances PLACED objects must be used, for example to talk to some external hardware such as a UART. In this case, it is often more efficient to create a local copy of the address by using an abbreviation, rather than referring repeatedly to the original object:

```
PORT OF INT uart :
PLACE uart AT #12345 :

PORT OF INT uart IS uart :  -- forces a local copy of the address
SEQ
  ...  use 'uart'
```

6.7 Block move

The transputer vector assignment instruction move is directly supported by the occam language. The vector assignment statement:

```
[65536] BYTE bigVec, otherVec :
[ bigVec FROM 0 FOR 65536 ] := [ otherVec FROM 0 FOR 65536 ]
```

compiles down to only 4 instructions:

```
ldl bigVec       -- assuming the vectors are abbreviated
ldl otherVec     -- locally
ldc 65536        -- this will be prefixed of course
move
```

A very fast vector initializer can be written using block moves:

```
PROC initBYTEvec ([]BYTE vec, VAL BYTE pattern)
  INT dest, transfer :
  SEQ
    transfer := 1
    dest     := transfer
    vec [0]  := pattern
    WHILE dest < (SIZE vec)
      SEQ
        IF
          (dest + transfer) > (SIZE vec)
            transfer := (SIZE vec) - dest
          TRUE
            SKIP
        [vec FROM dest FOR transfer] := [vec FROM 0 FOR transfer]
        dest     := dest     + transfer
        transfer := transfer + transfer
:
```

This performs a series of assignments of increasing length, initializing the first byte of the vector, then the next 2, then the next 4, 8, 16 etc., until it finishes the vector.

6.8 Use `TIMES`

The transputer has a fast (but unchecked) multiply instruction, which is accessed with the occam operator `TIMES`. An integer multiply on the IMS T414-20 takes over a microsecond — using `TIMES` this will take as many processor cycles as there are significant bits in the right-hand operand, plus 2 cycles overhead.

`a TIMES 4` takes only 6 cycles (300 ns). Therefore, when multiplying integers by small constants, use `TIMES`. Note that the IMS T800 Floating Point Transputer has a modified version of `TIMES` which optimally multiplies small negative integers, as have all later transputers.

The compiler uses this faster, unchecked, version of multiply for normal multiply operations if run-time error checking is disabled by means of the `U` command line switch.

6.9 Retyping — accelerating byte manipulation

Under certain circumstances retyping can be used to speed up byte manipulation. If it is necessary to frequently extract byte fields from a word, then retyping the word to a `BYTE` array is faster than shifting and masking. For example:

```
INT word :
[4] BYTE bWord RETYPES word :
SEQ
  ...  use bWord[0], bWord[1], bWord[2], bWord[3]
```

To access bits 16..23 in `word`, simply reference `bWord[2]`, which will generate:

```
ldlp  bWord   -- load base of bWord
adc   2       -- select byte 2
lb            -- and load it
```

To perform byte operations on large arrays it is worthwhile moving portions of the array to a local (on-chip) array; this is because a block move transfers words and is therefore much faster than accessing individual bytes from an off-chip array. For example:

```
[1024] INT vector :
[]BYTE bytevector RETYPES vector :

[16] BYTE local :
PLACE local IN WORKSPACE :
INT base :
SEQ
  base := 0
  SEQ i = 0 FOR 64
    SEQ
      local := [bytevector FROM base FOR 16]
      base := base + 16
      SEQ i = 0 FOR 16
        SEQ
          ...  use local[i] to access each byte
```

6.10 Scoping of variables

The compiler estimates the run-time count of the number of times each variable in a subroutine is accessed, using a heuristic which allows for repetition of loops, etc. It uses this information to place the most frequently used variables at the smallest workspace addresses; hence these variables can be accessed via smaller (and faster) instructions. The order of variable declarations has no major effect, unlike the situation in the D705B occam toolset; and the use of separate vectorspace makes the problem less acute anyway.

Variables should be scoped as locally as possible. The compiler uses the lexical scoping of variables to determine which variables are live (in use) at the same time, and which can be overlaid over each other. Hence localized scoping of variables can also reduce the total workspace requirement, thus helping to fit the total workspace into on-chip RAM.

`BYTE` and `BOOL` scalar variables are initialized at declaration by the compiler, to enable quick access as a local variable. Therefore it is not a good idea to declare them inside inner loops.

Note - Unfortunately this is not good programming practice. Declaring items at the scope within which they are required is more secure, preventing accidental modification and other programming errors. This conflict of good programming style and program efficiency is a feature of this compiler implementation, not the occam language. It is intended that future releases of the occam compiler will be more flexible.

6.11 Use the whole language

There are features of occam which are particularly suited to certain types of problems.

For example, when comparing an expression against a list of distinct constants, use a `CASE` statement rather than an `IF`. A compiler will attempt to make a `CASE` construct as fast as possible, assuming that all the values are equally possible, and may use a combination of techniques to select the correct branch. This compiler uses a combination of jump tables, binary searches, and explicit tests, depending on the relative values and density of the target values (i.e. whether there are any 'gaps').

An `IF` construct must be executed sequentially, evaluating each of the guards in turn. The first guard which is TRUE will be executed. Thus branches which are likely to be chosen frequently should be listed at the start of the `IF`.

Note that a combination of these two constructs may be the best solution where one value is particularly common, but where there may be many other possibilities:

```
VAL temp IS complicated.expression :
IF
  temp = frequent.value
    ...  process frequent.value
  TRUE
    CASE temp
      infrequent.value0
        ...  process infrequent.value0
      infrequent.value1
        ...  process infrequent.value1
      infrequent.value2
        ...  process infrequent.value2
      ...  etc
```

Note that replicated IFs are particularly suited to 'search' type lookups and comparisons:

```
BOOL FUNCTION equal.string(VAL []BYTE a, b)
  -- This returns TRUE if a = b
  BOOL result :
  VALOF
    IF
      (SIZE a) <> (SIZE b)
        result := FALSE
      IF i = 0 FOR SIZE a
        a[i] <> b[i]
          result := FALSE
      TRUE
        result := TRUE
    RESULT result
:
```

6.12 INLINE procedures and functions

This compiler allows you to write the keyword INLINE immediately before the keyword PROC or FUNCTION of a procedure or function declaration. The effect is that any call of that subroutine is expanded out as though the body were written in-line at the call site. This can be used to greatly improve program readability with no loss of performance. More importantly, this can improve performance by removing the overhead of the procedure or function call. It also allows the compiler to compile the body of the routine knowing the actual parameters, which provides further opportunities for optimization.

The programmer should be aware that inlining normally increases code size, and can cause problems because the calling procedure is then enlarged.

The current implementation does not permit the definition of the procedure or function which is to be inlined to exist in a different separately compiled unit to the caller. Instead, the declaration should be written in an include file, and #INCLUDEd by every source file which calls the routine.

6.13 Access to non–local variables

Non-local variables (i.e. those which are declared in an outer procedure or function) are accessed via a 'static link'. This will require one memory access for each level of nesting, every time that variable is accessed. It is possible to avoid doing this repeatedly, for example when a variable is used inside a loop, by creating a local abbreviation to that variable; this will create a pointer in the local workspace, and this local copy can be used inside the loop.

6.14 Access to formal parameters

All variable (non-VAL) parameters, and all VAL parameters which are either arrays, or longer than a word, are accessed via a pointer. If a parameter is accessed frequently, it may be worthwhile 'cache-ing' such a variable in a local variable:

```
PROC cumulative.sum(INT sum, VAL []INT array)
  INT local.sum :
  SEQ
    local.sum := sum    -- copy into a local variable
    SEQ i = 0 FOR SIZE array
      local.sum := local.sum + array[i]
    sum := local.sum    -- write back into the real variable
:
```

6.15 Pre–evaluate expressions

This technique is applicable to all programming languages. Any calculation which is to be performed repeatedly should be removed from any inner loops. If the calculation is relatively simple, it can be pre-evaluated by hand. Alternatively, at the beginning of the program a table can be initialized so that the values can be accessed quickly later.

As a trivial example, suppose a program requires frequent access to '*n* cubed' where *n* is always less than 100.

```
PROC init.cubes([]INT cubes)
  SEQ i = 0 FOR SIZE cubes
    cubes[i] := (i * i) * i
:
[100]INT cubes :
SEQ
  init.cubes(cubes)
  ...  other initialisation
  WHILE test  -- this is the 'inner loop'
    ...  use "cube[x]" instead of "(x * x) * x" here
```

6.16 Conditional expressions

Remember that by definition INT TRUE evaluates to 1 and INT FALSE to 0. This can be used to transform the following type of example

```
SEQ
  IF                -- this is slower
    test
      x := 1
    TRUE
      x := 0

  x := INT test  -- this is quicker
```

Note that some programmers consider the second form to be less readable, so the first could be left as a comment.

6.17 Array subscripts

Array subscripts of the form "`a[c]`" (where `c` is a constant) are evaluated most efficiently. However, if no range checks are required, and run-time error checking is disabled, "`a[e + c]`" (where `e` is any expression), is evaluated as quickly as "`a[e]`", as is "`a[c + e]`" and "`a[e - c]`".

6.18 `INT16`s

`INT16` values are not handled very efficiently on current 32-bit transputers. They should be converted to `INT` while being processed, and converted back to `INT16` to be stored, if they are really required.

If a mixed system of 16-bit and 32-bit transputers is being used, it may be more efficient to use `INT32`s as the portable communication values, since `INT32` values on a 16-bit processor are generally handled more efficiently than `INT16` values on a 32-bit processor. However, using `INT32`s will require twice as much data to be communicated and stored. The memory requirements may be particularly important on a 16-bit processor which only has 64K of addressable memory.

6.19 `ALT`s

Large, multi-way `ALT`s are relatively slow, since their time cost is proportional to the number of channels in the `ALT`. A technique known as fan-in can be used to enhance their speed.

Instead of, say, a 100-way `ALT`, it would be faster to use ten processes consisting of 10-way `ALT`s collecting the input, and passing that information to another process with another 10-way `ALT`. Each communication through the `ALT` then is processed by two 10-way `ALT`s instead of one 100-way `ALT`, and will be faster. Care must be taken, however, because using this model will change the synchronization properties of the program.

6.20 Use of `ASSERT()`

This compiler implements a predefined procedure `ASSERT(VAL BOOL test)`. If `test` is FALSE, and can be detected at compile time, this causes a compile-time error. If `test` can only be evaluated at run-time, the compiler will insert code to check that `test` is TRUE.

This can be used for various debugging tests, and to document and check various assumptions which have been made in the source code. If required, run-time **ASSERT** checks can be removed by using the compiler's **NA** command line option.

6.21 Transputer scheduler

This compiler implements a predefined procedure **RESCHEDULE ()**. This will place the current process to the back of the active process queue, and will work in either priority. **RESCHEDULE** should be used rather than relying on the implications of the transputer scheduling model. In some implementations of occam, the following "works", but the compiler is quite at liberty to optimize it out completely.

```
PAR
  SKIP
  SKIP
```

7 Summary

7.1 Optimizing for code size

Most of the optimizations described in this document will optimize for both space and time, but note:

- If an `INLINE` routine is called more than once, its body will be expanded multiple times.
- The use of separate vectorspace may cost code space because an extra parameter must be passed to every routine.

On the other hand, careful use of abbreviations can also reduce code size.

7.2 Removing run–time checks

Remember that run-time checks are not as costly as is sometimes thought.

The compiler's `K` switch removes any checks concerning array bounds violations. The `NA` switch removes all run-time checks of `ASSERT`. The `U` switch removes all code whose sole purpose is to detect errors (e.g. integer conversions, etc.). The `Y` switch disables interactive debugging and will speed up communication.

So to compile a program with no error checking, use:

```
oc myprog -h -na -u -y
```

7.3 Placing arrays in on–chip RAM

If the access time of a particular array is critical, and the workspace of a program fits into on-chip RAM, it will be useful to move that array into on-chip RAM instead of vectorspace, by inserting the following declaration after the array's declaration:

```
PLACE array.name IN WORKSPACE :
```

If the workspace does not already fit into on-chip RAM, it might be worth putting the vectorspace at the bottom of memory, instead of the program's workspace. Turn vectorspace off by default, by using the compiler's `V` option, but place the critical array into vectorspace explicitly:

```
PLACE array.name IN VECSPACE :
```

The `order.vs` configurer attribute can then be used to move the vectorspace to the bottom of memory. This attribute must be set in the `MAPPING` section of the program (see section 4.5).

```
MAPPING
  SET processor (order.vs := -1)
:
```

An alternative method which can be used for single processor programs which are not configured, is to use the collector's `S` switch (see section 4.6). The program should be declared with an extra parameter as follows:

```
#INCLUDE "hostio.inc"
PROC myprog(CHAN OF SP fs, ts, []INT mem, []INT fastmem)
  ...
:
```

The `fastmem` array is placed onto the on-chip RAM by the collector, and is allocated so that the space is not used for anything else. Hence this array can then be used for buffers, etc., which are required to be accessed quickly. For example, it could be used as the 'initializer' buffer for the `ClearScreen` subroutine described in section 6.4. The collector should be invoked as follows:

```
icollect myprog.lku -t -s 100
```

This will allocate 100 words at the bottom of memory, which are passed in as the parameter `fastmem`, and can be accessed more quickly than the rest of memory.

7.4 Placing code in on–chip RAM

Turn the critical subroutine into a separately compiled routine by passing all 'global' variables into it as parameters.

Add the following directive to the beginning of the subroutine:

```
#PRAGMA LINKAGE
```

This informs the linker to place the code for this subroutine in front of the rest of the code.

If the workspace requirement of the program is small and fits in on-chip RAM anyway, some of the code will be placed on-chip too. Since the linker has placed the critical routine at the start of the code section, this routine will be placed in on-chip RAM.

If the workspace requirement is large, it may be better to move the entire workspace off-chip, so that the code can be placed in on-chip RAM. This is done by setting the `order.ws` configurer attribute in the `MAPPING` section of the configuration program, which forces the workspace to reside at a higher address than the code. (see section 4.5).

```
MAPPING
  SET processor (order.ws := 10)
:
```

7.5 Building benchmarks

Compile the program in `HALT` error mode, and turn off all error checks with the `U` switch (see section 4.1), and the `NA` switch.

Disable interactive debugging with the `Y` switch. Use the linker, configurer and collector's `Y` switch too. (See section 2.5).

Also, disable virtual routing with the configurer's `NV` option. (See section 2.6).

Single processor programs may benefit from using the collector's `M` option to specify the memory size in advance, so that a simpler bootstrap may be used. (See section 4.6).

Experiment whether disabling vectorspace has any useful effect; this may be true if the workspace requirement is small anyway, as it commonly is with benchmarks. (See section 2.8).

Take particular care to ensure that the workspace is placed in on-chip RAM. Where possible, use linker section ordering to ensure that the 'inner loop' subroutines are also placed on-chip.

8 Maximizing multiprocessor performance

The following sections will describe how to obtain more performance from an array of transputers. However, only very general guidelines can be offered. Maximizing multiprocessor performance is still an area of active research, and any solution will tend to be specific to the problem at hand.

8.1 Maximizing link performance

The transputer links are autonomous DMA engines, capable of transferring data bidirectionally at up to 20 Mbits/sec. They are capable of these data rates without seriously degrading the performance of the processor. To achieve maximum link throughput from a multi transputer system the links and the processor should all be kept as busy as possible.

8.1.1 Decoupling communication and computation

To avoid the links waiting on the processor or the processor waiting on the links, **link communication should be decoupled from computation.**

For example, the following program is part of a pipeline, inputting data, applying a transformation to each data item, then outputting the transformed data:

```
PROC transform (CHAN OF protocol in, out)
  [dataSize] INT data :
  WHILE TRUE
    SEQ
      in  ? data
      applyTransform ( data )
      out ! data
:
```

If the channels `in` and `out` are transputer links, then the performance of the pipeline will be degraded. The `SEQ` construct is forcing the transputer to perform only one action at a time; it is either inputting, computing or outputting; it could be doing all three at once. Embedding the transformer between a pair of buffers will improve performance considerably:

```
PAR
  buffer ( in, a )
  transform ( a, b )
  buffer ( b, out )
```

The buffers are decoupling devices, allowing the processor to perform computation on one set of data, whilst concurrently inputting a new set, and outputting the previous set.

In this example the buffer processes will simply input data then output it. There is a transfer of data here which can be avoided, as all the data can be passed by reference:

```
[dataSize] INT a, b, c :
...  proc input
...  proc transform
...  proc output
SEQ
  input (a)               -- start-up sequence .. pull in data
  PAR
    input (b)             -- now transform that data
    transform (a)         -- and pull in more ...
  WHILE TRUE
    SEQ                   -- and from here on
      PAR                 -- the buffers pass round-robin
        input     (c)     -- between the inputter, transformer
        transform (b)     -- and outputter
        output    (a)
      PAR
        input     (a)
        transform (c)
        output    (b)
      PAR
        input     (b)
        transform (a)
        output    (c)
```

Instead of input and output operations transferring data between the processes, the processes transfer themselves between the data, each process cycling between the vectors `a`, `b` and `c` as the `PAR` statements close down and restart.

This is a special case, a data flow architecture where all communication and processing is synchronous — there is a lock-step **in, transform, out** sequence which allows this sequential overlay of computing and communication. This is not the case in many programs, where buffer processes are required.

8.1.2 Prioritization

Correct use of prioritization is important for most distributed programs communicating via links. If a message is transmitted to a transputer and requires through-routing, it is essential that the transputer input the message then output it with minimum delay — another transputer somewhere in the system could be held up, waiting for the message. In such cases it is important to **run the processes which use the links at high-priority.** There will tend to be more than one process talking to links, at most eight, and the `PRI PAR` statement allows only one process at each priority level. It is necessary to gather together all the link communication processes, unify them into a process with a `PAR` statement, and run this process at high-priority.

The program from above now becomes:

```
[dataSize] INT a, b, c :
...  proc input
...  proc transform
...  proc output
SEQ
  input (a)             -- start-up sequence .. pull in data
  PRI PAR
    input     (b)       -- now transform that data (HI-PRI)
    transform (a)       -- and pull in more ...
  WHILE TRUE
    SEQ                 -- and from here on
      PRI PAR           -- the buffers pass round-robin
        PAR
          input   (c) -- between the inputter, transformer
          output  (a)
        transform (b) -- and outputter
      PRI PAR
        PAR
          input   (a)
          output  (b)
        transform (c)
      PRI PAR
        PAR
          input   (b)
          output  (c)
        transform (a)
```

8.2 Large link transfers

Setting up a transfer down a link takes about about a microsecond (20 processor cycles), but once that transfer is started it will proceed autonomously from the processor, consuming typically 4 processor cycles every 4 microseconds (one memory read or write cycle per 32-bit word).

Keep messages as long as possible.

For example:

```
[300] INT data :
SEQ
  out ! some.data; 300; [ data FROM 0 FOR 300]
```

is far better than

```
[300] INT data :
SEQ
  out ! some.data; 300
  SEQ i = 0 FOR 300
    out ! data [i]
```

However, long link transfers increase latency when data must be through-routed. Some optimal message length will give the best compromise between overhead on setting up transfers, and overhead on through-routing.

9 Dynamic load balancing and processor farms

Processor farms are a general way of distributing problems which can be decomposed into smaller independent sub-problems. See INMOS Technical note 22 ("*Communication Process Computers*"), which was also supplied as chapter 4 of "*The Transputer Applications notebook - Architecture and Software*".

If implemented carefully, processor farms can give linear performance in multi transputer systems — that is ten processors will perform 10 times as well as one processor. Processor farms come into their own when solving problems where the amount of computation required for any given sub-problem is not constant.

For example, in the INMOS ray tracer one pixel may only require one traced ray to determine its color, but other pixels may require over a hundred.

Rather than give each processor say one tenth of the screen (assuming ten processors in the array), the screen is split into much smaller areas — in this case 8x8 pixels, giving a total of 4096 work packets for a 512x512 pixel screen. These are handed out piecewise to the farm. Each processor in the farm computes the colors of the pixels for that small area, and passes the pixels back, the pixel packet being an implicit request for another area of screen to be rendered.

Since work is only given to the farm on demand, load is balanced dynamically, with the whole system keeping itself as busy as possible. Buffer processes overlay data transfer with communication, reducing the communication overhead to zero, and the end-case latency of a processors farm implemented this way is far lower than in a statically load-balanced system.

The key to the processor farm is a valve process, allowing work packets into the farm only when there is an idle processor. The structure of this valve is:

```
PAR
  -- pump work unconditionally
  SEQ i = 0 FOR workPackets
    inject ! packet
  -- regulate flow of work into farm
  SEQ
    idle := processors
    WHILE running
      PRI ALT
        fromFarm ? results
          idle := idle + 1
        (idle > 0) & inject ? packet
          SEQ
            tofarm ! packet
            idle := idle - 1
```

where the crucial statement is the guarded **ALT**,

```
(idle > 0) & inject ? packet
```

only allowing work to pass from the pumper into the farm when there is an idle processor. The **ALT** is prioritized to accept results.

Index