Southampton's Portable Occam Compiler (SPOC)
Version 1.1
User Guide

P5404 GP MIMD

Our Ref: GPMIMD:013/O2C/GUIDE

Revision

Mark Debbage (md@uk.ac.srf.pact) Mark Hill (mbh@uk.ac.srf.pact)
Sean Wykes (smw@uk.ac.soton.ecs) Denis Nicole (dan@uk.ac.soton.ecs)

March 1, 1994

Contents

1 Introduction

The occam programming language provides a natural and concise syntax for specifying compositions of processes. occam has many advantages: it has primitive constructs supporting concurrency, it lends itself to distributed execution, it has a strong formal basis and it permits secure programming. occam is an excellent language for embedded systems and high-performance computing. To date however, there are only mature occam compilers for the transputer processor and this has restricted the exploitation of the language.

The motivation of this work was to develop a compilation system which would allow occam programs to be supported on industry-standard platforms, as typified by the UNIX work-station. Compilation is achieved by translating occam source into C and then using native C compilers to generate the target code. This approach allows a portable occam compilation system to be produced and allows back-end optimization to be performed by the C compiler in a manner optimized to the target processor. The implementation of an occam-to-C translator is not a trivial task since occam supports many features that have no direct equivalent in C. These features include concurrency, nested procedure definitions and a static variable chain.

This version of the Southampton Portable occam Compiler (SPOC) supports the occam 2 programming language as defined in [4]; implementation restrictions are documented in Section 4.2. The system generates ANSI C code that will run on a single target processor. The system has been developed with `gcc` version `2.4.5` running under SunOs version `4.1.3` on a Sun iPC Sparc-based work-station. The source-level debugging support requires the GNU debugger and has been tested with `gdb` version `4.8`. Currently the installation mechanism, the makefile and compilation scripts assume that the system is hosted by the UNIX operating system.

This document describes the use of the portable occam compilation system. It is assumed that SPOC has been correctly installed as described in [2]. The user is expected to be familiar with occam 2; the definitive language reference is [4] while a tutorial guide can be found in [5]. The remainder of this document describes how applications how compiled and debugged under SPOC, a list of features and restrictions, how to deal with problems, performance figures, possible future work and manual pages.

2 Compiling Applications

This section is a tutorial guide illustrating how occam programs can be compiled and executed using SPOC. This is achieved with reference to the example codes which are distributed with SPOC.

2.1 User Environment

This user guide assumes that SPOC has already been correctly installed for the target machine as described in [2]. The executable search path must contain the directory `$SPOC/bin/sun4` for the standard installation. For systems compiled to a different target the architecure specific component of this path must be changed appropriately. Furthermore, the user environment must declare an environment variable `$SPOC` which points to the base of the SPOC installation. Example C-shell commands for an installation in `/usr/local/spoc` are:

```
setenv SPOC /usr/local/spoc
setenv ARCH sun4
set path="$path $SPOC/bin/$ARCH"
```

By default the compiler will search for referenced modules in the current directory first and then in the directory `$SPOC/lib/$ARCH`. To search other directories (after the current directory but before the default SPOC directory), a variable `$OSEARCH` should be defined containing a list of directories

to search. Items in this list should be separated by a space or a colon, and they must have a trailing path separator (a / for UNIX). An example in C-shell syntax is:

```
setenv OSEARCH /home/users/me/mylibs/
```

2.2 Compiling A Simple Example Program

A simple occam example is shown in Figure 1. This program is distributed in the standard release as the file `$SPOC/examples/hello/hello.occ`. The code first opens an include file to reveal the definition of the server protocol. This is then used to print a string to the screen before terminating. The code uses a standard procedural interface; other possible top-level interfaces are described in the following section.

```
#INCLUDE "hostio.inc"
#USE "hostio.lib"
PROC main(CHAN OF SP fs,ts)
  so.write.string.nl(fs,ts,"hello world")
:
```

Figure 1: The inevitable 'hello-world' example.

The code can be compiled by issuing the following commands:

```
cd $SPOC/examples/hello
occ2c -C hello.occ
gcc -c -I$SPOC/lib/sun4 hello.c -o hello.o
gcc hello.o -o hello $SPOC/lib/sun4/libhostio.a $SPOC/lib/sun4/libconvert.a \
    $SPOC/lib/sun4/libIntrinsics.a -lm
```

This sequence produces an executable file, `hello`, which can be executed directly from the command-line. Such command sequences can be fully automated using a supplied tool. `omakef` examines the module structure of an occam program and produces a makefile that can be used to compile it. Firstly, `omakef` must be called with the name of the target executable to build:

```
omakef hello
```

This produces a makefile called `hello.mkf` which can be built using:

```
make -f hello.mkf
```

If `omakef` is invoked with a `-z` option then it will display referenced files as they are encountered. Any other command-line options are passed into the `occ2c` compilation commands in the generated makefile. If the debugging option `-g` is supplied this is propagated to the C compilations as well as the occam ones. This allows default compilation modes to be overridden without manual customization of the makefile. Any parameters to `omakef` containing = are assumed to be definitions of makefile variables. These are inserted in the generated makefile after the standard `omakef` makefile header but before any rules. This mechanism allows other makefile defaults to be changed. This technique is commonly used to change the arguments to the C compiler. A common example is to invoke `omakef` with `CCOPTIONS+=-O2` which appends an optimization option to the usual C compilation parameters.

2.3 Compiling Single-File Programs

All processes in occam source files must be contained within an appropriate procedure or function interface. The top-level file of any compilation must contain a single procedure with one of the

predefined procedural interfaces defined in Figure 2. These interfaces describe the parameters that the system can pass into an occam program and are compatible with the entry-points used by Inmos Toolsets. The actual entry symbol (`main` in the figure) is not checked and may be any valid occam identifier.

```
PROC main () :
PROC main ([]INT memory) :
PROC main (CHAN OF SP fs, ts) :
PROC main (CHAN OF SP fs, ts, []INT memory) :
```

Figure 2: Top-level procedural interfaces.

The parameters represent the following quantities:

- `CHAN OF SP fs` is the connection from the host server.
- `CHAN OF SP ts` is the connection to the host server.
- `[]INT memory` is an array of available memory space.‘

The server channels are not actually required by the system as distributed as it makes use of the standard C library to perform host activity. However, users with access to the Inmos library sources can compile them up under SPOC to obtain full coverage of the `hostio`, `streamio` and `string` libraries, as shown in the installation guide [2]. The actual amount of memory passed into the program is sized by a default value (1000 words) but can be overridden by the first command-line argument.

2.4 Compiling Multi-Module Programs

The modularization of occam 2 programs is not defined by the language. The approach taken here is compatible with that used by Inmos Toolsets and therefore uses `#INCLUDE` and `#USE` compiler directives.

Source files can be included into an occam source using the `#INCLUDE` directive. This is given a filename parameter in quotes locating a textual file. This file inserted into the source code at the current level of indentation. This mechanism is typically used to share protocol definitions and constant values between program modules.

Object files can be referenced from an occam source using the `#USE` directive. This is given a filename parameter in quotes locating a file generated by running the compiler on the referenced module. This file contains declarations of the entry-points into the referenced module allowing the compiler to construct and check parameters correctly. This mechanism is also used to size workspace requirements across module boundaries.

Files referenced in this manner are located using the standard compiler search mechanism described in Section 2.1. The local directory is searched first, then any directories listed in the `$OSEARCH` path followed by the default compiler directory. SPOC's extension conventions are defined in Appendix A.1. These extensions allow compiled programs from SPOC and from the Inmos Toolsets to coexist in the same directory.

The command-line steps to compile modules are complicated due to the need to produce procedural interface definitions as well as object files. This is hidden from the user by the `omakef` utility. `omakef` searches through file references allowing makefiles of hierarchical multi-module occam programs to be generated automatically.

The `omakef` system is organized so that if it can find source for a referenced module (i.e. the source exists on the search path) then the module is compiled up as an integral part of the program. If no source can be found then the module is accessed as a library unit and no rules are generated to compile the library itself. Library units should be organized so that SPOC libraries are located

through SPOC's search mechanisms (the `OSEARCH` path and the SPOC default). This ensures that SPOC library code is distinct from Inmos library code even though both use the same extension (`.lib`).

Building a multi-module occam program under SPOC is no more difficult than a single-file program. Simply use `omakef` to generate a makefile for the executable of the top-level occam file, and invoke `make` with the generated makefile.

2.5 Compiling Library Units

A library unit is typically a number of include files and a collection of procedures and functions. These may be distributed across any number of source files. A library unit differs from a module part of a compilation because it is compiled separately from target programs.

Once again the command-line complexity of generating library units is hidden by `omakef`. A single-file library called `simple.occ` can be compiled by invoking:

```
omakef simple.lib
make -f simple.mkf
```

The resultant library consists of an interface definition `simple.lib` and a library object file `simple.a`. These should be installed on the SPOC search path by copying the files to a suitable directory. Additionally, the UNIX naming convention for library files should be adhered to by renaming `simple.a` to `libsimple.a`.

For a multi-file library, a top-level file should be written referencing each source file in turn through `#USE`s. A top-level library called `multi.occ` can be compiled by invoking:

```
omakef multi.lib
make -f multi.mkf
```

As before the resultant files, `multi.lib` and `multi.a`, must be installed on the SPOC search path.

If a library implementation references a library that does not form part of that library then an implicit library dependency occurs. This is typified by the `hostio` library which implicitly references the `convert` library. Such dependencies must be propagated up to the makefile for the user program and this is achieved through files with a `.liu` extension which record inter-library dependencies. If such a dependency occurs while processing a library, `omakef` generates an appropriate `liu` file and this should be installed on the SPOC search path alongside the associated library.

2.6 Calling C Code

The compiler provides a C code insertion mechanism. This is accessed through the `#C` and `#H` compiler directives. The `#C` directive can be used wherever an occam process is allowed, the `#H` directive is restricted to the outermost level of the file (to introduce C `#include`s for example). The directives pass any text following them straight through to the C compiler. occam variables can be accessed by prefixing a `$` to the occam symbol name.

For scalar types, this mechanism yields the value of the occam symbol; pointers to scalar occam symbols can be taken by preceding the `$` with an `&`. For array types declared locally or passed in by reference, this mechanism yields the first entry in the array. This can be converted to a pointer to the array by prefixing a `&` to the occam symbol reference. For array types passed in by value, this mechanism yields a pointer to the array automatically. Note that occam arrays are passed as C pointers rather than C arrays so that C array subscription can only access the lowest dimension. For other dimensions it is necessary to locate the extent of lower dimensions and to use explicit

pointer arithmetic. If you wish to manipulate multidimensional occam arrays from C it is strongly recommended that indexing is done through occam abbreviations prior to the C process.

2.7 Customising omakef

The makefile generation tool is a virtual necessity for using the system as producing makefiles even for fairly simple applications is error-prone and time-consuming. The tool is highly customisable to ensure that it is *never* necessary to modify the generated makefiles by hand. *Any changes made by hand to generated makefiles will be lost next time the tool is run.*

The tool can be configured using the following techniques:

2.7.1 Supplying options to omakef

Any switches on the `omakef` command line which appear after the target filename, are passed to the `occ2c` command line. In addition if the occam debugging option `-g` is supplied this is propagated to the C compilations. Any parameters containing equals are assumed to be make macro assignments and inserted in the makefile after the defaults.

2.7.2 Modifying the template file

The makefile is contructed by performing straightforward textual substitutions from a template file, concatenating the modified templates and introducing additional compilation dependent macros. The template file `Omakef.tpl` is searched for along the following path: the current working directory, left to right through the directories on the `OSEARCH` path and finally the architecture-specific library directory (`$SPOC/lib/$ARCH`). This allows the scope of `omakef` customisations to be:

System wide – by modifying the template file in the architecture-specific `library` directory of the SPOC installation.

User specific – by placing a local directory containing the template file at the start of the OSEARCH path.

Application specific – by modifying the template file copied to the directory containing the application.

All occurences of `FILE` in a template are substituted with the appropriate target filename. Values for `FILE-DEP` and `FILE-LDD` macros are produced automatically by `omakef`. The first lists all the occam files on which a `FILE` is dependent. The second lists all the binaries on which it is dependent.

3 Debugging Applications

All the occam source level debugging facilities are dependent on the GNU Debugger (the system has actually been developed using `gdb` version 4.8). No changes have been made to the debugger itself: all support is done through appropriate changes to the generated C, a `gdb` command script (`occam.gdb`) and a simple shell script (`odebug`).

This section takes the form of a tutorial, all the examples codes required are distributed in the directory `examples/buggy`. If you are sharing an installation you are advised to take a local copy of the directory before proceeding.

Start by producing an executable from `buggy.occ` (see Figure 3). As you might have guessed there are some errors in this code so you might as well start by turning on the debugging option. Type `omakef buggy -g` as `-g` is the compiler option which generates the additional code required

```
 1 -- For use in conjunction with the SPOC User Guide
 2 -- See section on source-level Debugging
 3 -- Debugging example code 1
 4
 5 PROC jim()
 6   PROC fred([]INT array,VAL INT a,VAL INT b)
 7     SEQ
 8       SEQ i = 0 FOR SIZE array
 9         IF
10           (i \ 2)=0
11             array[i] := a
12   :
13   INT a :
14   VAL b IS 4 :
15   SEQ
16     a := 2
17     [5]INT A :
18     INT j :
19     SEQ
20       fred(A,a,b)
21       j := 0
22       WHILE TRUE
23         SEQ
24           A[j] := j
25           j := j + 1
26 :
```

Figure 3: The 'buggy' example.

for debugging. Note that any options supplied on the `omakef` command line are introduced as additional arguments to the compiler in the generated makefile rules. This should generate a makefile `buggy.mkf` which can then be used to produce the executable `buggy`.

On execution of the code you should obtain the following message:

```
All IF branches failed.
ERROR FLAG raised on line 11 of file buggy.occ
Terminating application.
```

Note that this message would have been generated even if debug had been turned off. To run the debugger type:

```
odebug buggy
```

This brings up a help page and places you at the first executable line of the program:

```
*****************************************************
* SPOD : Southampton's Portable Occam Debugger     **
* M. Debbage, M. Hill, S. Wykes                    **
* University Of Southampton, ESPRIT GPMIMD P5404   **
* Shell for standard GNU gdb utility               **
* Usual GNU licence applies                        **
*****************************************************

Some useful gdb commands:
break            Sets a break point e.g.
                 "break <c_fn>" Use olistsubs to determine <c_fn> from occam equiv.
                 "break file.occ:12" Sets bp at line 12 in module file.occ
step             Executes occam line by line
cont             Continues execution after breakpoint
list             Lists code around current line of execution

Occam debugging command set:
rerun            Restarts execution of program
oids             Displays values of a list of occam identifiers
oall             Displays values of all occam variables in scope
ostep            Steps through the occam program displaying selected variables
odown            Displays values of all occam variables declared in callee's stack frame
oup              Displays values of all occam variables declared in caller's stack frame
oframe           Displays values of all occam variables declared in current stack frame
oshow            Repeats most recent occam stack dump
olistsubs        Lists occam subroutine names and their C equivalents
onext            Proceeds to start of next scheduled process

Scheduler () at buggy.c:98
98              ret = (tFuncPtr)CURTASK->Func (CURTASK->FP);
P_jim_buggy (FP=0x1edc8) at buggy.occ:13
13        INT a :
spod>
```

At the `spod>` prompt type `cont`, the program will proceed to the point at which the error flag was set and then return to a `spod>` prompt. Type `list` to view the code around the point of error and then `oall` which will list the value of all `occam` identifiers in scope at that point.

This reveals that the replicator index `i` has reached a value for which no `IF` conditional is true. To avoid this a second clause is appended to the end of the `IF`, the resulting program is distributed in `lessbuggy.occ`. Type `quit` to exit the debugger then generate a makefile for `lessbuggy.occ`, run `make` and execute `lessbuggy`. This time the following error should occur:

```
Error- Range check (0<=5<5) failed
ERROR FLAG raised on line 26 of file lessbuggy.occ
Terminating application.
```

To investigate this failure enter the debugger and set a breakpoint at the start of the `WHILE` loop on line 24 (the command is `break 24`). Continue execution to that point (`cont`) and then `list` the code. We are clearly interested in the values of `A` and `j` so type `set ID="A j"` and then `oids`. Enter `ostep` to see how the variables change, and then hit return a few times (which repeats the previous command), then try `step`ing a few times. Finally, `cont`inue to the error and type `oids`. This reveals that the value of `j` has now been incremented beyond the end of the array.

The final code unit, `debugged.occ` introduces a test to exit the loop as `j` reaches the arrays upper bound. Compile and execute this example and you should find the program does absolutely nothing!

4 System Features

This section highlights the features and limitations of this SPOC release. Unexpected application behaviour should be checked against these properties first before assuming that the system is erroneous.

This section highlights the current features and limitations of SPOC.

4.1 Implementation Features

SPOC supports the occam 2 programming language as defined in [4]. Implementation restrictions are described in 4.2. Other features of SPOC are:

- Generated C code is ANSI-C compliant.
- C code generated is fully portable between target platforms
- Efficient portable support for concurrency.
- Optimized code generation for atomic code units, including use of automatic variables wherever possible..
- Support for *alien* language procedure and function calls (C, F77 and NAG F90)
- Code insertion mechanism for in-line C code.
- All processes in occam source files must be contained within an appropriate procedure or function interface. Modules may contain any number of procedures and/or functions with arbitrary parameter lists. The entry point to a program is a single procedure with one of the predefined interfaces defined in the SPOC user-guide [3].
- Channel tables are supported.
- Extensive checks on array subscription, array ranges and conversions.
- Detected errors can stop process, terminate application or be ignored.
- Support for multi-module programs and library units using `#USE` and `#INCLUDE`.
- Automatic sizing of workspace across procedure and module boundaries.
- Support for the standard occam 2 libraries defined in [4] (`Intrinsics`, `snglmath` and `dblmath`).
- Partial support for the Inmos-standard occam 2 libraries (`hostio`, `string` and `convert`).
- Optional support for simulated iserver which runs as separate Unix process and talks SP protocol via a socket to the occam program. This gives full hostio and streamio support but requires that the user has access to proprietary Inmos library sources.
- Automatic makefile generation using `omakef`.
- SPOC compilations and Inmos Toolset compilations can coexist in the same directory.
- Source-level occam debugging through `gdb`.
- Compiler implements alias and usage checking.
- Coarse grained (60/100 ticks a second) *non-premptive* timer support for delayed input and selection.
- Conventional compilation environment allows use of software engineering tools such as SCCS and RCS.

4.2 Implementation Restrictions

- This release generates code for a single target processor.
- No support for `PLACE` or `PORT`.
- There is no overflow checking during expression evaluation apart from type conversions.
- This release of SPOC has only rudimentary support for the real-time features of occam 2. This may be addressed but will require support from the host operating system. The current code generation strategy does not support pre-emption. This is currently sufficient since there is no external communication. Two levels of priority are supported but the restrictions ensure that switches between the two are at controlled scheduling points.
- Incomplete run-time support for a configuration level (no `PLACED` or `PROCESSOR`).
- The `streamio` library has not been reimplemented; many `hostio` operations are not yet implemented. However, the simulated server mechanism allows those users with Inmos Occam Toolsets to compile up and use the standard libraries.
- Parallel usage checking only supports arrays subscripted by simple expressions. These expressions are limited to linear functions of replicator indices. The usage checker cannot analyse subscipts invovling modulo operations. These restrictions can be worked around by demoting usage violations to a warning or by turning off usage checking.
- The data-type `INT64` is only supported if the target compiler supports 64-bit `long long`s.
- The extent of `PAR` replicators must be a compile-time constant.
- There is currently no support in `omakef` for ordinary C or Fortran modules.

4.3 Other Differences To Transputer occam

This section documents areas in which the behaviour of SPOC is different to transputer occam.

- The transputer is little-endian whereas the endian-ness of SPOC is determined by its target processor. Hence little-endian assumptions made through `RETYPES` abbreviations may not hold for the SPOC system using the default code-generation mode. A compile-time option causes the compiler to generate true target independant code, performing any required retypes at run-time. This can be expensive, particularly when retyping unbound arrays, as temporary storage must be allocated at run-time. The compiler must be informed of the endianness assumptions of the source occam, in order to correctly perform `RETYPE`d constant folding, but has support for all combinations of host / target endianness, without the user needing to know the endianness of the compiler.
- Access to host facilities (such as screen output, keyboard input and file access) is achieved through the conventional C mechanisms. The server channels `fs` and `ts` are still passed around like in Inmos systems but they are not synchronized upon. Hence, these channels cannot be used to synchronize accesses to the host and the results may not be identical to a transputer implementation. Furthermore, explicit communications over these channels will deadlock the involved processes since there is no server listening to them. To get the same I/O semantics as the Inmos Toolset it is necessary to have access to the Inmos library sources and compile them under SPOC, the process is detailed in the Installation Guide..
- In the standard release all host interactions are implemented through the conventional C run-time system and will block all occam processes until the call completes. In particular occam processes cannot operate concurrently with keyboard input. Removal of this restruction requires compilation of the Inmos libraries and use of the simulated iserver facility.

5 Compilation Details

This section gives a brief overview of the components of the occam compilation system. This information is not needed by the everyday user of the system but may be read for interest or to help narrowing down compiler problems.

5.1 User Environment

The SPOC system demands that an environment variable `$SPOC` is initialized to point to the SPOC installation. Furthermore, the tools provided by SPOC are expected to be on the executables path. If files are not found correctly then ensure that the instructions in the installation guide[2] were followed precisely. Details of how to recompile the SPOC system are also given in the installation guide; it is assumed here that a functional compiler has been successfully installed.

5.2 Makefile Generation

The makefile generation tool is relatively dumb but is extremely flexible. The command sequences to issue partial and complete compilations of modules are contained in a textual template file. The tool `omakef` simply walks through an occam source tree producing a makefile entry for each item of source that is to be compiled. This is complicated slightly by the details of producing suitable indirect file lists and by implicit references between modules. Any command-line switches after the target filename are added to all `occ2c` invocations. In addition a `-g` option will also be propagated to all C compilations.

`omakef` has been tested with GNU make version 3.68 and the UNIX make distributed with SunOs version 4.1.3.

5.3 Occam-to-C Translation

The occam-to-C translation process can be split into a number of phases and it is usually possible to narrow problems down to a particular phase. Firstly, the command-line option `-v` will force the compiler to indicate its phase of operation on the standard output. The option `-zl` can be used to get the compiler to only scan (lex) the input file, while the option `-zx` will generate lexical debug. The option `-zd` produces parsing debug, `-zt -ps` will display the abstract syntax tree after parsing and `-po` will regenerate an occam source from the tree. The option `-pp` can be used to prevent the compiler from performing parallel usage checking, although this will also disable some code generation optimizations. A `-zt` option will show the abstract syntax tree after transformation while `-po` will regenerate an occam source from the transformed tree. The option `-o -` directs the generated C code to standard output while `-pc` will inhibit the generation of any C code.

5.3.1 Lexing And Parsing

The lexing and parsing capabilities of the occam-to-C translator have been extensively tested through many hand-crafted examples and several large occam programs. If the occam-to-C front-end fails to lex and parse a valid occam 2 code then the problem should be reported. The problem code should of course be narrowed down to the smallest amount of code that demonstrates the bug and, if possible, the code should be tried with an Inmos compiler.

5.3.2 Checking

The type-checking of the compiler should detect any type violations in the code. Alias and usage checking of arrays are currently restricted to simple expressions involving constants and linear functions of replicator indices. Alias and usage checking of scalar variables is complete.

5.3.3 Tree Transformation

The compiler undertakes a considerable amount of work to attribute the abstract syntax tree with useful information. These values are commonly synthesized up the tree or inherited down the tree using attribute grammars. The values can be observed by displaying the tree, although familiarity with the compiler internals will be required to interpret their meaning.

The compiler performs abstract syntax tree transformations of occam constructs that are difficult to map into C. All processes within a `PAR` or replicated `PAR` are pushed out into procedures, while `VALOF` blocks in expressions are pushed out into functions. These new routines are retained in the nearest declaration list in order that scope is not violated. Table accesses in expressions are also pushed out into new declarations in a similar manner. The effect of code transformations are most easily be seen by enabling occam code generation (`-po`).

5.3.4 Code Generation

The largest component of the occam-to-C translator is the back-end that generates ANSI C code from the transformed abstract syntax tree. All occam variables are held in a single static data structure to allow static chaining to work and to ensure that occam local variables are retained over descheduling points. Code with descheduling points (arising from communications for example) is coded up inside case statements switched by a simulated instruction pointer. This allows the run-time system to maneuver the control flow between arbitrary descheduling as required by the scheduler.

The code is generated in five main parts:

- The first phase of the code generator defines structure names for all internal functions and procedures. These names are defined before any internal data structures are generated to prevent circular definitions[1]. This phase also defines the contents of structures for external functions and procedures. These contents are exactly defined by the procedural interface to the external and its workspace requirement.
- The second phase of the code generator produces data structures to hold the occam locals for each procedure. These are built up into a single static structure containing the entire occam memory map for the program. The compiler generates uses union structures to allow workspace to be reused wherever appropriate; for example, variables declared within subprocesses of a sequential construct will share state.
- The third phase of the code generator produces any C statics that are required by occam declarations outside of the scope of any procedure. Special care is needed for tables at this scoping level since C static tables are required to have 'obviously-constant' values.
- The fourth phase of the code generator produces the actual C code implementing the occam program. Processes containing descheduling points are coded up within case statements as described above, while other processes (termed atomic) are optimized to use the natural C implementation. Procedure and function instances write parameters directly into the workspace of the called routine.
- The final phase is only of concern for modules. It produces code allowing the top-level procedure and function interfaces to be revealed along with their channel usage and workspace requirements.

5.3.5 Mapping Of Data Types

This section describes the mapping of occam data-types into the equivalent C. The default primitive type mappings are shown in Figure 4. Additionally, the occam compiler can be recompiled up to map `INT64`s into `INT32`s if the C compiler does not support the `long long` type.

[1] The parent of a particular procedure needs to reference that procedure to instance state for it, while any particular procedure may potentially need to reference its parent to access non-locals.

occam type	C type
INT	long int
INT16	short
INT32	long int
INT64	long long
BYTE	unsigned char
REAL32	float
REAL64	double
BOOL	char

Figure 4: Default primitive type mappings.

occam array types are declared using C arrays of the appropriate primitive type. Abbreviations of occam array types (including array parameters) are handled as pointers to primitive types since C arrays do not have the generality of occam arrays. All array subscription is therefore implemented through explicit pointer arithmetic with the extent of array dimensions taken account of explicitly. Array accesses and abbreviations are checked unless this feature is disabled.

5.4 The Run-Time System

The run-time system is organized around the scheduler and the scheduling queues. During the execution of a program, the scheduler takes the highest priority process that is waiting to execute. This process is activated by calling the associated function with its saved workspace pointer. When this process reaches a descheduling point or terminates, control is returned to the scheduler.

When a process makes a call to a non-atomic procedure, control is returned to the scheduler and the scheduler then calls the new procedure. This scheme ensures that all processes that can deschedule can immediately drop back to the scheduler without penalty. Completion of the called procedure invokes the scheduler again before returning to the caller. Calls to atomic processes are optimized using a conventional C-style call.

Routines supporting process creation, process joining, channel communication and channel alternation are built in terms of the scheduling operations. These are written as macros and functions; if GNU C is used the functions are in-lined as an optimization.

5.5 C Compilation

Compilation of the ANSI C source files generated by the translator should be straightforward. The directory `$SPOC/lib/$ARCH` must be added onto the source file search path of the compiler. This is of course achieved automatically if `omakef` is used. By default occam line numbers are propagated into the C source so that errors can be quickly traced back to the occam source. The philosophy of the whole compilation system is that the C source is considered as a portable assembly language, and can be disregarded by the end-user of the system.

5.6 Linking

The linking of object modules into programs uses the conventional linker provided by the underlying operating system. For UNIX systems this is the program `ld`. Similarly, modules are grouped together into libraries using the appropriate operating system program; this is `ar` for UNIX. Link-time problems are usually limited to problems resolving symbol references. These should not occur if `omakef` is used since this tracks down all occam file references and generates an appropriate linker command-line. Furthermore, all occam references are checked at occam compilation-time and these should all link correctly. The C run-time library and the C maths library are always linked in by `omakef`-generated makefiles.

5.7 Execution

The execution of a SPOC-generated program is achieved simply by quoting its name. The behaviour of the code on the detection of an error can be switched between three modes by compiler options:

- The default mode is to terminate the application with an appropriate error message.
- Alternatively, the errant process can be halted.
- Alternatively, the error can be ignored.

If the code reaches a deadlocked state or a completed state then the scheduling queue is empty and the program will terminate. If the code is compiled up with debugging enabled then the debugging techniques described in Section 3 can be employed.

5.8 When Things Go Wrong

The Southampton Portable occam Compiler is a large intricate software system. This version is an alpha release and also the first time that the compiler has been distributed beyond the implementation team. It is therefore inevitable that things will go wrong, especially for large dusty occam codes.

The first course of action is to reduce the encountered problem to the smallest piece of code that demonstrates the errant behaviour. This should be checked against the list of restrictions to see if it requires a feature that has simply not been implemented yet. If possible, the code should be tested with an Inmos occam compiler to check its validity.

Any bug that can be narrowed down to a fault within the SPOC distribution itself should be reported to the implementation team via:

E-mail	`smw@uk.ac.soton.ecs`
Paper-mail	Sean Wykes
	Room 3053, Mountbatten Building
	Dept. of Electronics and Computer Science
	University of Southampton
	Highfield, Southampton SO9 5NH
Fax	+44 703 593045

All bug reports must clearly state the version number. This is given by running the compiler with a `-V` option or inspecting the banner of the `README` in the installation's base directory. Our intention is to supply support where possible for GP-MIMD project members within the duration of the GP-MIMD project. Any other support is entirely at the discretion of the implementation team.

6 Advanced architectural and OS-related issues

The SPOC translator has a number of command-line options which influence the both the strategy used for code-generation and the assumptions made by the translator. Similarly, a number of compile-time `#define`s select or disable run-time features and optimisations, some of which are system dependent.

6.1 Processor Endian-ness

By default, SPOC assumes that the occam source code has been written to assume the endian-ness of the processor for which SPOC was compiled. Thus, if SPOC is run on a sun4, the default source

endian-ness is assumed to be little-endian. This assumption is necessary to enable constant folding and propagation of `RETYPE`ed constants to be performed correctly. The `-me` option reverses this endian-ness assumption, allowing , for example, `occam` code written with big-endian retypes, to compile correctly. A further option, `-mi` causes SPOC to generate code which, as far as retypes are concerned, is fully target architecture independent, containing inline endian-ness reversal code. The processor endianness is evaluated at C-compile time, and the retypes are only performed where necessary. Note that the use of this mode involves a certain run-time processing overhead.

6.2 Cross compilation

It is possible to cross-compile `occam` applications using SPOC. The application generation process is split into two distinct stages, translation and compilation. Under normal conditions, `Omakef` generates makefiles containing rules for both stages, the use of the `-t` option (before the root filename on the `Omakef` command line) will generate a makefile containing rule only rules for translation. Similarly, the `-c` option generates a makefile contains only compilation rules. The filename extension for the makefile will change from the default `.mkf` to `.mkt` for translation and `.mkc` for compilation respectively.

6.3 Timer support

Run-time support is included for `TIMER ? AFTER .` statements and has two modes of operation. The default (system independent) mode causes the inputting process to be rescheduled immediately, but placed at the end of the process queues, allowing other processes a chance to execute. The compile time option `#define USESIGTIMER`, enables the second mode of operation which emulates the transputer timer operation. A time-ordered queue of processes waiting on timer is kept, processes are appended to this queue when they perform a timer `AFTER` input, which requires a delay. A signalling `ALARM` is set up during the code initialisation; this is continuously enabled and will reschedule waiting processes once their delay-until time has passed. This has currently been tested under SunOS, and relies upon operating system support for `SIGLARM`.

6.4 ISERVE and socket communications

Using sockets for iserver support, optional signal use to allow async ISERVER comms.

6.5 PLACED PAR and multi-processor support

Work is currently in progress to support multi-processor operation and a configuration level.

6.6 Porting to other architectures / operating systems

There are a number of things to be considered when porting SPOC to other architectures. The SPOC translater is written largely in ANSI-C, so porting of the translator should be a relatively painless process. The main problems which may be encountered will be in the runtime system.

- The standard uni-processor run-time system is system independent, and supports efficient scheduling and communication
- The timer `AFTER` code relies upon SunOS support for a signalling timer. The existing code should port straighforwardly to another OS which supports a similar construct
- The ISERVER files `hostc.c, filec.c, iserver.c serverc.c` contain support for a number of host systems (Sun3, Sun4, PC, VAX) and operating systems (SunOs4, MSDOS, VMS, HELIOS). Changes may be required to support additional architectures and operating systems

- The use of asynchronous socket communications has currently only been tested on a Sun4 system. Operating support for signalling socket comms is required for this facility to work

7 Performance

The remainder of this section considers the performance of the translator on three example source codes. Two of these codes are relatively large and place a demanding test on the compilation system. The target platforms were:

- 25 MHz T800 using SPOC and Inmos `icc`.
- 25 MHz T800 using SPOC without run-time checks and optimizing Inmos `icc` (`-O2`).
- 25 MHz Sparc IPC using SPOC and GNU `gcc`.
- 25 MHz Sparc IPC using SPOC without run-time checks and optimizing GNU `gcc` (`-O2`)..
- 40 MHz SuperSparc CS-2 using SPOC and GNU `gcc`.
- 40 MHz SuperSparc CS-2 using SPOC without run-time checks and optimizing GNU `gcc` (`-O2`)..

The reference platform was a 25MHz T800 transputer and the Inmos occam Toolset (D7205).

In addition to these codes, SPOC has successfully compiled the Inmos Toolset library source, numerous small code fragments and a number of large occam applications including a Forth system written in occam and a 60000 line INMOS simulation code.

7.1 Highly Concurrent Example

This example was donated by Peter Welch of the University of Kent. The program consists of four simple processes interacting through channels to produce an output stream of counting integer numbers. Each iteration of the test involves four communications which give four context switches with the SPOC code generation strategy. The amount of calculation is very small, so that the test gives a very good measure of the cost of context switching. In many senses, this is a worst case code for the translator.

The occam source contains 50 lines (1 KByte) of text including the code to show the result. When translated to C this becomes 330 lines or 6KBytes excluding debug information and initialisation code. The execution time per iteration of the example is shown in Figure 5.

Platform	Iteration Time (μs)
T800 (Inmos occam)	12
T800	169
T800 (optimized)	168
Sparc	60
Sparc (optimized)	36
SuperSparc	17
SuperSparc (optimized)	9

Figure 5: Performance for highly concurrent example.

7.2 Network Simulation

The second code was a network simulation. This program also featured large amounts of concurrency but within the framework of a real application. The code consisted of 3000 lines of occam (98 KBytes) split over six modules. The generated C code contained 9350 lines of source (233 KBytes). The results in Figure 6 show the total time to complete the simulation.

Platform	Simulation Time (sec)
T800 (Inmos occam)	260
T800	1863
T800 (optimized)	1838
Sparc	822
Sparc (optimized)	507
SuperSparc	255
SuperSparc (optimized)	173

Figure 6: Performance for network simulation.

7.3 Computational Fluid Dynamics (CFD)

The third application was a simulation of fluid flow through a pipe. The code was provided by Nick Floros of the University of Southampton and was developed in the ESPRIT PUMA project. The program was originally formulated for 16 processors using the Virtual Channel Router[1]. The top-level configuration file was converted back to an occam source module for the purposes of the test. The code exhibits some concurrency but also contains a large amount of floating point arithmetic.

The original code contained 9300 lines of occam (300KBytes) and the translated C code had 25000 lines of C (950KBytes). The results in Figure 7 show the total time to complete the simulation. There are no results for this code on the SuperSparc.

Platform	Execution Time (sec)
T800 (Inmos occam)	900
T800	3030
T800 (optimized)	2850
Sparc	1300
Sparc (optimized)	640

Figure 7: Performance for a CFD simulation.

8 Future Work

This section is a repository of ideas for future work. It should not be read as a statement of intent to actually do all of these things!

- Fix all restrictions described in Section 4.2.
- Use user-level threads library to give pre-emptive behaviour between priorities.
- Support a configuration layer with support for distributed execution.
- Support more target machines and operating systems; principally PCs with DOS extenders and/or Windows NT.
- Complete implementation of standard occam libraries.
- Enhanced parallel usage checking, including modulo support in replicator expressions
- Transform any redundant parallelism into sequential code where possible.
- Improved debugging support, including inspection of task queue and state of inactive processes.
- Support data-types of occam 3.
- Support all of occam 3.
- Fix all compiler bugs.

9 Conclusion

Southampton's Portable occam compiler allows occam 2 programs to be developed, compiled and executed on any platform that supports an ANSI-standard C compiler. It also allows occam programs to be source-level debugged using the GNU debugger. The vast proportion of the language is supported and the system has been successfully tested with several large applications. Current work consists of compiler validation using the INMOS occam test suite and bug-fixes.

Preliminary performance figures illustrates that a reasonable level of efficiency can be achieved even with processors with relatively high context switch times. In fact, results from the SuperSparc indicate that it can exceed the performance of T800 transputers even with highly concurrent code. There is still considerable scope for back-end optimization and the performances currently achieved are therefore only a starting point.

The compiler has been implemented with a relatively meagre amount of man-power. It represents about 12 man-months split between the 3 members of the implementation team. This has been possible due to the use of the GMD-Karlsruhe Compiler Toolset which allows compilers to be developed at a high-level of abstraction.

10 Acknowledgements

The authors would like to acknowledge contributions from the following individuals and organisations:

William Pugh (`pugh@cs.umd.edu`) et al. at the Department of Computer Science, **University of Maryland**, College Park for the Omega test. They have produced both papers and public domain software to implement the test which determines whether there is an integer solution to an arbitrary set of linear equalities and inequalities. This is applied to perform range, alias and parallel usage checking to array accesses performed with arbitrary linear replicator expressions. More information about the test can be found in the [6]. Their software is available via anonymous ftp from `ftp.cs.umd.edu` in directory `pub/omega` and is released with the following disclaimer:

> The implementation of the Omega test and extensions to tiny have been done by a number of people at the University of Maryland:
>
> William Pugh
> Dave Wonnacott
> Udayan Borkar
> Wayne Kelly
> Jerry Sobieski
> Vadim Maslov
>
> This software is public domain, and no legal restrictions of any kind whatsoever are placed on uses of it). You may do whatever you want with it, and no guarantees of any kind are made about its performance or lack of errors. You can copy it, use it, incorporate it or even sell it. We request that any research or products that make use of this software acknowledge that use and that you keep us informed of your use.
>
> Please send mail to omega@cs.umd.edu if you wish to be added to a mailing list for people interesting in using this software. We will notify people on the mail list of bug fixes and new releases.
>
> Also send mail to omega@cs.umd.edu if you have any trouble installing the software, bug reports or questions.
>
> Our work on this software has been supported by NSF grants CCR-8908900 and CCR-9157384 and by a Packard Fellowship, as well as being based on Michael Wolfe's original implementation of tiny.

Josef Grosch and associates at **GMD Karlsruhe** for his compiler construction tool box. The following tools have been used to help construct the compiler:

bnf - Parser generator

cg - Command grammar tool for generating abstract syntax trees and attribute grammars.

puma - Tree Pattern matching tool for transformation and code generation.

We used Version 9208 of the tools whose Copyright is retained by GMD (1989, 1990, 1991, 1992). Direct requests, comments, questions, and error reports should be addressed to:

> Josef Grosch GMD Forschungsstelle Vincenz-Priessnitz-Str. 1 D-7500 Karlsruhe 1
> Phone: +721-662226 Email: `grosch@karlsruhe.gmd.de`

The compiler construction tool box is distributed via anonymous ftp in compressed tar format from `ftp.gmd.de` in the directory `gmd/cocktail`.

David Beckett and **Peter Welch** at the **University of Kent at Canterbury**, the former for his manual pages to the standard Occam libraries, the latter for suggesting the scheduling benchmark distributed as part of the examples (examples/comstime). The manual pages for the libraries are copyright David Beckett (1993), University of Kent at Canterbury and are also available via anonymous ftp from `unix.hensa.ac.uk` in `pub/parallel/documents/ukc`.

The lexical analyser is loosely based on that developed by Peter Polkinghorne. His disclaimer is reproduced here:

> This work is in the public domain. It was written by Peter Polkinghorne in 1986 & 1989 at GEC Hirst Research Centre, Wembley, England. No liability is accepted or warranty given by the Author, still less my employers.

Tony Debling at **Inmos Limited** has given kind permission for us to distribute the standard occam Toolset include files `hostio.inc` and `mathvals.inc`. Copyright is retained by Inmos.

References

[1] Debbage, M., Hill, M. B., and Nicole, D. A. The Virtual Channel Router. *Transputer Communications 1*, 1 (Aug. 1993), 3–18.

[2] Debbage, M., Hill, M. B., Nicole, D. A., and Wykes, S. M. Southampton's Portable Occam Compiler (SPOC), version 0.9a, installation. GP-MIMD Technical Note GP-MIMD:012/O2C/INSTALL Revision 0.1, University Of Southampton, Oct. 1993.

[3] Debbage, M., Hill, M. B., Nicole, D. A., and Wykes, S. M. Southampton's Portable Occam Compiler (SPOC), version 0.9a, user guide. GP-MIMD Technical Note GP-MIMD:013/O2C/GUIDE Revision 0.1, University Of Southampton, Oct. 1993.

[4] Inmos Ltd. *occam 2 Reference Manual.* Bristol, United Kingdom, 1988.

[5] Pountain, D., and May, D. *A tutorial guide to occam programming.* BSP Professional Books, 1987.

[6] Pugh, W. A practical algorithm for exact array dependence analysis. *Commun. ACM 35*, 8 (Aug. 1992), 102–114.

A SPOC Conventions

A.1 Filename Extensions

The `omakef` utility imposes extension conventions on the filenames generated by its makefiles. These extensions have been chosen so that the extensions of the generated files are distinct from Inmos conventions. This allows SPOC and Inmos occam Toolset compilations to coexist in a directory. The conventions are shown in Table 8.

File Type	SPOC Extension	Inmos Extension
Source occam file	`.occ`	`.occ`
Source include file	`.inc`	`.inc`
Generated C code	`.c`	n/a
Executable to generate prototype information	`.x`	n/a
Object prototype information	`.t`	`.tco` (or similar)
Object code	`.o`	`.tco` (or similar)
Library prototype information	`.lib`	`.lib`
Library code	`.a`	`.lib`
Executable	(no extension)	`.btl` (or similar)
Makefile	`.mkf`	`.mak`

Figure 8: Filename Extensions.

Under `omakef` prototype information is held in `.t` files for objects and `.lib files` for libraries. The utility distinguishes between objects and libraries according to whether the source code can be found. When a module is referenced by a `#USE` and the source code is found on the search path, then that module is treated as an object. This means that it is compiled up in the generated makefile and its prototypes are generated in a file with a `.t` extension. If the source code is not found then that module is treated as a library unit. The module is therefore not compiled up in the makefile.

The Inmos Toolsets place object code and their associated prototypes in a single file. This was inappropriate for SPOC since its object file format is a UNIX standard. The only clash between the Inmos extension scheme and the SPOC scheme is for library files. This is satisfactory however, since library files will be held in distinct directories accessed through `ISEARCH` for the Inmos system and `OSEARCH` for SPOC. Additionally, `omakef` will not generate `.lib` files as part of a program compilation since it will use the `.t` default for modules with source code.

The recommended way to reference modules from `#USE` directives is to simply quite the module name without any extension, since this allows the source code to remain portable. Both `omakef` and the Inmos equivalent (`imakef`) will then add appropriate extensions as appropriate. Note that `imakef` automatically propagates the compilation target and compilation mode through `#USE` directives formed in this manner. Separately-compiled libraries should always be accessed with a `.lib` extension.

An alternative approach is to quote the module name with an extension. The `omakef` tool uses `.t` extensions for modules and will also replace any extension it does not recognize with a `.t` extension. This allows programs using the Inmos `.tco` (or equivalent) conventions to compile correctly.

B Procedural Interface to C, Fortran 77 and Fortran 90

SPOC has the ability to interface to C, Fortran 77 (via f2c) and Fortran 90 (NAG) [?], at the procedural level, as well as allowing inline C code within the occam source itself. Currently, the user is required to construct a prototype file, containing occam-style definitions for the external routines.

B.1 Protofile file definition

A prototype `.t` or `.lib` file consists of a sequence of colon terminated prototype definitions, one per line, each defining the interface to an external procedure in the following format:

```
LANGUAGE PROC procedure_name (args...) module_name :
LANGUAGE typelist FUNCTION proc_name (args...) module_name :
```

where *LANGUAGE* is defined in figure 1.

LANGUAGE	PROC	FUNCTION
EXTERN_ATOMIC	Atomic occam procedure	Atomic occam function
EXTERN_OCCAM	Non-atomic occam procedure	Invalid
EXTERN_C	(void) C procedure	Invalid
EXTERN_F90	NAG F90 module procedure	Invalid
EXTERN_F77	f2c procedure	Invalid

Table 1: Extern Prototype keyword definitions

Note: occam prototype definitions are generated automatically by SPOC, the information is included here for completeness.

args... is an optional comma separated list of argument types. The data-type mappings and valid combinations for each language will now be covered in more detail.

B.2 Interface to C procedures

The procedure name as defined in the prototype directly specifies the called procedure name, with the following exceptions; any periods in the name will be converted into underscores (as the prototype name must be a valid occam identifier) and the module name (if non-blank) will be postpended to the name, separated by an underscore.

Mapping of primitive occam-types to C has already been shown in Figure 4. The modifiers in figure 2 affect these primitive types.

occam argument	C argument type	comments
`TYPE`	type *	pointer to primitive type
`VAL TYPE`	const type	scalar argument
`[]TYPE`	type *	array bounds unchecked
`[n]TYPE`	type[n]	array bounds checked
`[m][n]TYPE`	type[m][n]	array bounds checked
`VAL []TYPE`	const type *	array bounds unchecked
`VAL [n]TYPE`	const type[n]	array bounds checked
`VAL [m]...[n]TYPE`	const type[m]...[n]	array bounds checked

Table 2: Prototype argument translations

Note: There is a difference in representation between occam character strings and C strings, occam strings are represented as an array of characters, with seperate length information where required, whereas C-strings are null terminated. Therefore when passing occam strings to C, a null terminating character must be explicitly added.

B.3 Interface to NAG F90 procedures

The procedure name and module name as defined in the prototype are used to construct a procedure name. The Fortran 90 code should be compiled up as a module, an example is shown in Figure 9.

```
MODULE mod
CONTAINS
  SUBROUTINE fred(a,m,b)

  INTEGER a      ! Not modified
  REAL m(10)     ! Not modified
  REAL b(10,:)
  .
  .
  END SUBROUTINE
  SUBROUTINE jim(...

END MODULE
```

Figure 9: Example Fortran 90 module

NAG Fortran 90 procedure arguments are always passed by reference, and the mappings shown in Figure 10 are available (the kind values are specified in a module included with the f90 installation). Fortran array parameters of fixed shape should be specified within the prototype, noting that the order of the dimensions should be reversed (Fortran arrays are column major). Fortran array parameters of assumed shape are correctly handled, with the extents of any unbound dimensions being calculated and passed in at runtime, via *dope vectors*. If a parameter is constant (ie. unmodified within the fortran), it should be specified as `VAL` within the prototype, in order for the occam type checking to work correctly.

occam type	F90 type
`INT`	INTEGER
`INT16`	INTEGER(KIND=int16)
`INT32`	INTEGER
`INT64`	Not available
`[]BYTE`	CHARACTER(sk)
`REAL32`	REAL
`REAL64`	REAL(KIND=double)
`BOOL`	LOGICAL(KIND=byte)

Figure 10: Data type mappings - occam to NAG f90.

A prototype for the example shown in Figure 9 would be:

```
EXTERN_F90 PROC fred(VAL INT, VAL [10]REAL32,[][10]REAL32) mod :
```

B.4 Interface to F2C Fortran77 procedures

The procedure name is used with the addition of a postpended underscore. All parameters are passed by reference and the datatype mappings are the same for F90. The only exception being that array extents must be specified at compile time.

NAME

occ2c – Translator from Occam Source into C

SYNOPSIS

occ2c [options] file

Options: [–dghHilsuvw] [–o <file>] [–m[acefilmnprvw]] [–e[s|t|c] [[–t<number>] [–p[acopts]] [–z[cdelnqstvx]]

DESCRIPTION

This manual page documents versions 0.9b and later of **occ2c,** the Occam to C translator of the **SPOC (Southampton Portable Occam Compilation) system.** The standard action of the command

occ2c <file>.occ

is to take the occam program contained in <*file*>*.occ* and produce an ANSI C version of the program in <*file*>*.c* The default settings of all the various options can be seen with the **-h** option.

The translation system internally consists of several phases:

Lexical Analysis and Parsing
Type Checking
Attribute evaluation
Code Transformation
Code Generation.

TRANSLATION AND COMPILATION

An Occam program consists of a number of code modules and libraries. The outer-level code module (main module) may reference any number of library units, which can be sub-modules of the same program, or actual libraries.

ENVIRONMENT

occ2c requires a number of predefined header and template files to be available, both for internal usage, and for include and library files. The defaults for these files are located through the *SPOC* environment variable which should be directed towards the root directory of the compiler installation. These files can be superceded either by placing them in the current working directory or by adding an *OSEARCH* environment variable containing a sequence of space-seperated paths, each terminated by a slash. The search order is current working directory, left to right through the directies listed in *OSEARCH* and finally default locations relative to the *SPOC* environment variable.

GENERAL OPTIONS

–d Enable inline C directives in the Occam source. These are generally used in library files for access to host i/o facilities. There are two directives,

#H <line_of_C_code>

which is only allowed at the outermost level (typically for include C header files) and

#C <line_of_C_code>

which is valid anywhere an occam process is allowed. Simple occam variables can be used within the C code by prefixing them with a

–o <file>

Outputs the generated C code to the *<file>*, the filename – can be used direct the code to standard out.

–s This option forks off an SP server as a separate Unix process which talks SP protocol via a socket to the SP channels passed in to the top of an application. This gives full inmos iserver compatibility but degrades IO performance and introduces a dependence on the proprietary Inmos libraries. When used versions of the spserver.lib and the associated proprietary Inmos libraries MUST APPEAR on the OSEARCH path.

–t<number>
Sets tab to be equivalent to *<number>* spaces, the default is 8. If no number is supplied then the option toggles whether tabs are allowed in the occam source at all.

–u Generates warnings for usage violations, rather than errors. This allows translation to continue to code generation, despite the presence of a range of errors including parallel usage errors (such as writing to a variable in multiple parallel threads) and alias errors (such as writing to an abbreviated variable/array).

–v Generate verbose translation information, including the translator version and build date, and messages describing the various translation phases as they are performed.

–w Suppress warning messages. No warning messages will be displayed, only errors.

CODE GENERATION OPTIONS

–e[s|t|c] **Select error mode. Occam programs handle** ways, either ignoring the error, stopping the erroneous process, or terminating the application. The error mode is selected by a suffix letter:
s – Stop process on error
t – Terminate application on error (the default)
c – Continue on error (ignore errors)

–g Enable source level Occam debugging.

–l Enable/Disable generation of library units (as opposed to a main module).

–m Change code generation mode, suffixed by any combination of:
a - disable Array checks
c - disable Conversion checks
f - enable calls to NAG F90
m - enable coMmunication checks
r - disable Range checks
n - disable generatation of occam line Numbers
e - does all RETYPE constant folding assuming a target of OPPOSITE endianness to the machine running the compiler.
i - generate target independent code to perform RETYPES at run-time
l - enable INT64 support by using GNU C's *long long* data type.
w - use default INT size of 16-bit. BEWARE code generated this way CANNOT be used with any modules/libraries compiled for 32-bits.
p - enable PLACED PAR support
v - enable free variable access from within PLACED PAR

INTERNAL TRANSLATOR DEBUGGING OPTIONS

–i Enable / Disable automatic inclusion of standard Occam intrinsic function library. This option is enabled by default, but must be disabled to compile the intrinsic library itself.

–p Control internal compilation phases. Suffix by any combination of:
- a - disable c output Attribute calculations
- c - disable c code generation
- o - enable occam code generation (to standard out)
- p - disable parallel usage Attribute calculations
- t - disable typecheck Attribute calculations
- s - disable code Simplification/transformation

–z Enable/Disable internal debugging options, suffix by any combination of:
- c - Enable CodeGenerator Debugging
- d - Debug parser
- e - Sort error messages (enabled by default)
- l - Lex only
- n - output Occam line numbers as comments within generated C
- q - Query syntax tree
- s - show Symbol table
- t - show abstract syntax Tree
- v - show variable declaration details
- x - leXical debug

NAME

odebug – Source-level debugger for Occam programs

SYNOPSIS

odebug file

DESCRIPTION

This manual page documents the source-level debugger of the **SPOC (Southampton Portable Occam Compilation) system. odebug** is a Unix shell script file, which executes the **GNU gdb** debugger preparing it (with a predefined set of macro commands) for debugging Occam programs. When the script is run it must be supplied with the name of an executable file which has been produced by running **occ2c** with a **–g** option and then compiling the generated C using **gcc** with a **–g** option. The debugger runs the program, executing the C preamble and then stops at the first line of occam, displaying the source for that line.

A tutorial describing use of the debugger can be found in the *SPOC User Guide*.

The most useful commands available are summarised below, some are standard **gdb** commands, others are defined in the script **occam.gdb** which is distributed with the system.

break [*file***:**]*line*
Set a breakpoint at line *line* (in *file*).

break [*file***:**]*function*
Set a breakpoint at *function* (in *file*). The specified function name must be that of the C function generated for the occam subroutine, the mappings are given by the **olistsubs** command.

step Executes occam line by line.

cont Continues execution until next breakpoint

list [[*file***:**]*line*]
Lists code around current line of execution or at *line* (in *file*) if those optiona are supplied.

rerun Restarts execution of occam program

oall Displays values of all occam variables in scope

oids Displays values of a list of occam identifiers

A space separated identifier list should be placed in ID using the set command before this command is called. For example:

set ID="a b fred"

oids

ostep Steps through the occam program and displaying the new values of the currently selected occam variables.

oframe Displays values of all occam variables declared in current stack frame.

oup Displays values of all occam variables declared in caller's stack frame. This can be called repeatedly to progress further away from the current stack frame.

odown Displays values of all occam variables declared in callee's stack frame. This can be called

repeatedly to return back towards the current stack frame.

oshow Repeats most recent occam variable dump.

olistsubs
Lists all occam subroutines in the module and gives the C function names which they map to. This is useful for placing breakpoints on entry to a specific routine.

onext Steps through the execution of the program stopping at the start of each freshly scheduled process.

BUGS

Bugs?

NAME

omakef – Makefile generator

SYNOPSIS

omakef [omakef-options] file [occ2c-options] [env=string]..

DESCRIPTION

Makefile generator for **SPOC (Southampton Portable Occam Compilation)** system. The compiler support tool omakef generates makefiles to allow occam applications to be compiled. The command sequences to issue partial and complete compilations of modules are contained in a textual template file. In principle, the tool simply walks through an occam source tree producing a makefile entry for each item of source that is to be compiled. The tool has been tested with GNU make version 3.68 and the unix make distributed with SunOs version 4.1.3.

COMMAND LINE

file Name of target file to be produced by the generated makefile. The tool examines the extension of this target file to select the actions required to generate it. Currently supported extensions are:

<*no extension*> Generate makefile for executable.

.lib Generate makefile for library prototype file.

The filename of the makefile is derived by stripping any extension from *file* and appending the extension .I .mkf , except when options .I -t or .I -c are specified, when either a .I .mkt or .I .mkc are appended.

OMAKEF OPTIONS

–t Generate a makefile containing only rules involving the translation from occam to C.

–c Generate a makefile containing only rules involving the compilation and linking of generated C files.

–z This option switches on debug information from omakef showing which files are referenced by which module.

OCC2C OPTIONS

–g Enables on source-level debugging by propagating the option to all occam-to-C translations and all C compilations.

Any command-line option containing an ‘equals’ character
is propagated to the generated makefile between the header definition section and the rules. This allows the default compilation options to be overridden or appended.

Any other option : These are propagated directly to the
occam-to-C translator.

BUGS