Southampton's Portable Occam Compiler (SPOC)

Mark Debbage Mark Hill Sean Wykes Denis Nicole

Department of Electronics and Computer Science
University of Southampton

February 17, 1994

Abstract

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 supports secure programming. occam is an excellent language for embedded systems and high-performance computing. To date however, the only mature occam compilers are for the transputer processor and this has restricted the exploitation of the language.

This paper describes a compilation system which allows 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.

1 Introduction

The Southampton Portable occam Compiler (SPOC) supports the occam 2 programming language as defined in [7]. The system generates ANSI-C code, implementing efficient portable concurrency running on a single target processor. Compiler features include support for multi-module programs and libraries, extensive compile- and run- time checks on array subscription, array ranges and conversions, optimized code generation for atomic code units and source-level debugging via the GNU `gdb` debugger.

The development of SPOC has been supported by the European Commission Esprit Project *GPMIMD* P5404. It is available free of charge and may be of interest to computer manufacturers requiring portability between transputer and non-transputer systems, commercial developers having existing occam applications, educational institutions wishing to teach the fundamentals of concurrent programming, the occam3 development group and anybody with an interest in either the occam language or concurrent computation.

1.1 The occam language

Although occam is an excellent language for exploiting tightly coupled arrays of transputers, it is just as applicable to the programming of far more weakly coupled systems such as workstation clusters and even sequential processors.

The language's clean syntax and clear semantics alone make it highly appropriate for such areas as teaching and formal verification[1]. It is also applicable to programming software systems which are most accurately modelled by a collection of interacting processes; particular examples include simulators and client/server systems.

The formal basis of the language and it's clearly defined semantics allow a significant quantity of compile time checks to be performed. Comprehensive type checking, array access bound checks, alias and parallel usage checking can be done by the compiler; these are augmented with run-time checks on, for example, array accesses.

The combination of type- and usage- secure language constructs with the explicit parallelism and synchronous communication abilities, make occam an excellent choice as a parallel harness for other high-level languages, allowing the interface between parallel modules and threads to be cleanly specified. This approach has significant advantages over the addition of parallel constructs to existing languages.

Unfortunately, the language's uptake by the computing community has been stifled by the lack of compilers for any architecture other than the transputer. This symbiosis is unhealthy for both the transputer and for the occam language. The extended use of the language is hampered because transputers are not readily accessible to the vast majority of the computer users and because organisations are unwilling to transfer their programming effort to a language which is only implemented by a single company running on one architectural series. Increasing the popularity of occam will also heighten the importance of the transputer as it provides an extremely efficient implementation of the language. The first transputer products were, in fact, designed specifically to run occam.

1.2 C as a Target Language

Central to the occam compilation system is the occam to C translation tool. Input to the translator is a convential occam 2 code module, as defined by the language's reference manual[7] but extended to include the standard Inmos mechanisms for multi-module support (the `#INCLUDE` and `#USE` pragmas). The output from the tool is target-independent ANSI C. This requires the efficient handling of parallelism through a mechanism which is only dependent on ANSI C features. A description of the efficient portable support for concurrency will be described in Section 2.

There are several advantages in targetting C as the intermediate language for the compiler. The most important of these is portability; the provision of an ANSI C compiler is now a prerequisite for any industry standard architecture. This approach therefore allows occam programs to be run on platforms as diverse as PCs, workstations, message-passing multi-processors such as the CS2[10] and shared memory multi-processors such as the SICS DDM [6]. This wide platform base makes the system as appropriate for teaching as it is for high-performance computing.

To aid in its dissemination, the package is freely distributed via anonymous ftp from the University of Southampton[1] and from the Transputer and Parallel Computing Archives at the University of Kent[2].

Other advantages of producing C are the ability to directly map occam primitive data types onto C types, the ease of code generation, which allows rapid implementation of the system, the ability to exploit both the general and architecture specific optimisation phases of modern C compilers, and access to standard langauge support tools such as linkers and debuggers. The system support's interactive occam level debugging using the Gnu tool `gdb`. The facilities available from occam include breakpointing, line-by-line execution and variable inspection.

There are some minor disadvantages with taking this approach, but they are far outweighed by the portability gained from using C.

- The C is produced relatively quickly but is much larger than the occam module from which it is generated (typically a factor of 4-5). The expansion is caused by a combination of the declaration and manipulation of large data structures (in which the occam program is executed), the inline inclusion of information taken from include files, debug data and the more verbose nature of the generated C. This may cause a problem when compiling large occam programs on low specification machines. As a consequence of this, and the incorporation of the C run-time library, the resultant binaries are significantly larger than would be expected for a native occam compiler.
- The implementation of parallelism and occam's block scope rules makes it difficult for an optimising compiler to perform inter-procedural and global optimisations. The key causes of this are the use of call by function pointer in the scheduler and the tortuous control flow during execution of communicating processes.
- Support for occam specific debugging facilites such as the concurrency and scoping rules must be handled by maintining additional state at run-time and providing routines which can be entered from the debuggger to inspect their contents. This means that the integration with `gdb` is not seamless; this is, however, still preferable to having to implement a debugger for every target architecture.

1.3 The GMD Compiler Construction Toolkit

The GMD Toolkit is a very comprehensive package of specification tools developed at Karlsruhe to support the construction of compilers[5]. They are available via anonymous ftp[3]. Documentation is acceptable although it is targeted towards those already familiar with compiler construction.

[1] `ftp.ecs.soton.ac.uk(/pub/occam/spoc1.1)`
[2] `unix.hensa.ac.uk(/pub/parallel/software/compilers/occam/spoc1.1)`
[3] `ftp.gmd.de` in the directory `/gmd/cocktail`

The key tools are:

Rex - A scanner generator which supports the generation of highly efficient scanners from a specification based on regular expressions. The tool can generate either a C or Modula-2 code unit with a single entry point (`GetToken`) which reads characters from standard input until one of the specified regular expressions is recognized. The tool provides a level of functionality similar to that of *Lex*[9] but is considerably faster. The toolkit supports conversion of source rex from lex.

Lalr - A parser generator for grammars which are LALR(1). Grammars are specified using EBNF, while the parsers semantics are introduced by annotating the rules with actions written in the target language. Typically these will be calls to the node construction primitives exposed by modules using the *Ast* tool. Lalr is similar to Yacc[8] but is considerably faster and provides much better error recovery.

Ast - A tool which accepts a formal specification of all node structures (including child and attribute types) required within an abstract syntax tree and generates routines for tree construction, checking, traversing and viewing. This tool is extremely useful as its hierarchical description and strict typing of nodes, coupled with an automatically generated node type checker, dramatically reduces the possibility of errors in the construction of the syntax tree.

Ag - This tool accepts a set of attribute grammars for the specified syntax tree and generates a tree traversal routine which propagates the attribute information around the tree as required by the attribute grammar.

Puma - A tool for performing the transformation and translation of abstract syntax trees. The tool accepts two inputs, the first is a "view" of the decorated syntax tree (identifying which attributes are of interest), the second associates node structures with semantic actions. A code module is generated which attempts to pattern match the input tree against the specified node structures. When a match occurs the semantic actions associated with the rule are executed.

Reuse - A collection of useful reusable code modules, for performing set operations, identifier hashing, allocation and use of dynamically resizable arrays, string storage, etc).

2 Efficient Portable Support for Concurrency

occam programs are written by composing processes together using constructors. The principal feature that distinguishes occam from most other languages is that occam processes can be composed in parallel just as easily as they can be composed sequentially. Since C does not provide language support for concurrent execution, the mapping of occam's concurrency requires the use of special techniques. Furthermore, occam encourages the use of parallelism to structure programs, aiding code modularization and reuse. This means that occam programs often exhibit large amounts of concurrency and efficient support is vitally important.

An occam process is an encapsulation of data and calculation with channels to allow it to communicate with other processes. Implementations of occam need mechanisms to support the independent execution of processes subject to their synchronizations. Since each process has a context it is necessary to provide independent workspace areas for processes so that the state of a process can be retained over synchronization points. The execution of an occam program on a single processor is achieved by multi-tasking, the serialization of the 'concurrent' processes through the processing unit. This multi-tasking can be considered as non-local flow control; when a process deschedules on a synchronization point the control flows out of that process, through the scheduler, and into another process that is ready to execute.

This flow of control cannot in general be determined at compile time and is therefore performed by a run-time system which maintains a scheduling queue of processes that are ready to execute (at each priority level) and switches between them in an appropriate manner.

2.1 Implementation Options

The traditional way to achieve concurrency from C is through a library interface. UNIX operating systems provide support for heavyweight processes which execute in separate address spaces. Thread libraries, such as Posix threads[12], provide an interface to user-level threads which execute in the same user address space. Since inter-process memory protection is inappropriate for occam, full-blown UNIX processes are not required and user-level threads are sufficient. Unfortunately, the overheads of both interfaces are often high due to the large amounts of processor context that need to be switched. Furthermore, both of these approaches require the use of libraries which are not part of the ANSI C standard and thus portability would be compromised.

A well-known alternative approach is to implement concurrency using the library routines `setjmp` and `longjmp`. These provide non-local control flow between different call frames on the stack. The approach allocates the workspace for processes from the stack and then switches between them using these routines. This technique abuses the ANSI C interface since it uses non-local jumps in both stack directions. ANSI C only specifies that non-local control flow back up the stack returns the state correctly. Returning up the stack has the effect of deallocating all lower stack frames and no assumptions can be made as to the value. The effect of jumping down the stack is, in principle, undefined but it works if the memory retains the deallocated stack frames. On some architectures, however, the deallocation of the stack frame can deallocate a physical page of memory and there is no guarantee that the same page will be recovered by a new allocation. The `setjmp` and `getjmp` trick will not work on such architectures. Furthermore, the use of this technique requires that the stack requirements of processes are estimated. It is only possible to generate constant-sized stack frames from ANSI C; the non-ANSI `alloca` library is required for dynamic-size stack allocation. In practice, the cost of using `setjmp` and `longjmp` is similar to user-level threads libraries since similar amounts of processor state are saved in each approach. On the Sparc processor in particular, the overhead incurred with either user-level threads or `setjmp` and `longjump` techniques is significant due to the system call involved. In addition, non-local jumps are normally used for exception recovery and not as part of normal control flow. Compilers behave very conservatively with regard to non-local jumps, and their use can prevent many standard compiler optimizations.

A common approach in simulation is to use an interpreter to handle the concurrency. This can be applied to the occam translator by compiling the concurrent program structure into a static table of actions which break the program down into atomic sections of code separated by synchronization points. The scheduler selects a pending process and executes it by invoking the actions recorded in this static table. occam programs are typically written, however, with large amounts of communication and this often occurs at the deepest parts of occam hierarchies. Atomic code sections therefore tend to be very short and the interpretative approach breaks programs into many tiny functions. This makes the generated code hard to relate back to the original program and debugging is extremely difficult.

The chosen approach for SPOC was to implement the non-local control flow that results from concurrency using ordinary C constructs. The scheduler is a C function that examines scheduling queues and invokes the next process to run. Once within a process a mechanism must be provided to return control to the scheduler at a descheduling point. The scheduler then chooses the next process to execute and routes control flow back to the continuation point of the selected process. This kind of control flow suggests that the C code must be organized so that any descheduling point is close to the scheduler. To achieve this, any procedure call to a procedure containing descheduling points is implemented by returning to the scheduler and letting the scheduler make the call. The effect is that all descheduling points are only one return away from the scheduler.

The state of procedures with descheduling points cannot be held on the C stack since returning to the scheduler would deallocate this state. Instead, variables must be held statically in order that their value is retained over descheduling points. Since calls to procedures with descheduling points are made from the scheduler a further mechanism is required to pass parameters. This is achieved by the caller writing the parameters directly into the occam stack frame of the callee. The advantage of this approach is that the parameters can be treated just like locals of the callee, and can be accessed non-locally by procedures within the lexical scope of the callee.

The scheduling control flow is realized by implementing the body of each procedure as a case statement. The case statement is switched by an instruction label to denote which atomic code section is to execute next. These labels are represented by a set of contiguous positive integers so that the C compiler can implement the case by efficient means such as a jump table or binary search. Every atomic code section commences with a case label and terminates with a descheduling point, a simulated procedure call or a simulated return. Control flow constructs containing descheduling points (such as sequential loops, if statements, while statements, case statements and alt statements) are built using the enclosing C case statement. This ensures the process can be restarted at any synchronization points without losing the context of any local control regardless of the level of control nesting.

The parallel components of a `PAR` constructor or replicator are moved into seperate procedures by occam to occam transformations. This simplifies the design of the back end and reduces the width of generated case constructs.

Processes that do not contain descheduling points are termed atomic and can be optimized significantly. Variables declared within atomic processes can be held on the C stack since there are no descheduling points to hold their values over. Atomic processes are implemented using conventional C constructs such as if, while, for and switch statements in the obvious manner. Procedures that are constructed from atomic processes and functions (which are atomic by definition) are themselves atomic routines. These can be implemented using a normal C call since there is no descheduling point to keep close to the scheduler. These are important optimizations since they allow the execution of atomic code to be unhindered by static variable access, unusual call sequences and contorted control flow. Existing C compilers are able to optimize atomic code sections well since they are very similar to conventional C codes.

2.2 *Run-Time System*

The run-time system to handle concurrency consists of the scheduler and routines to implement occam's parallel constructors and communication primitives. The scheduler has already been described. It is essentially the outermost procedure in the system which simply invokes the next process with the highest priority from the scheduling queues. The rest of the concurrency support is implemented as macros and functions. Most of the work is accomplished within functions; these can be compiled in-line if supported by the C compiler. The macros are used to simplify the output of the code generator and to make the generated C code more readable.

The lowest level of the run-time system implements the descheduling and rescheduling of processes. A process deschedules by removing itself from the front of the scheduling queue and returning to the scheduler. A process can reschedule another by adding that process to the appropriate scheduling queue. If the rescheduled process is of a higher priority than the executing process then control is returned back to the scheduler to allow the higher priority process to continue.

The run-time system provides routines to start processes and to end processes. A process is started by initializing a task data structure and appending it to the appropriate scheduling queue. A process ends by forming a join with all its sibling processes; when the join is complete the parent process continues. The task structure is used to hold process state when processes are descheduled. This includes the function which the process is currently executing, the instruction label of the continuation point within that function, the process's static link, the process's priority and any synchronization information that needs to be held over descheduling points. It also includes a reference back to its parent's task to allow the join to be implemented.

The run-time system provides input and output communication primitives. The first party to reach a communication finds that the channel is empty, records its details into the channel and deschedules by returning to the scheduler. When the other party reaches that communication it finds a non-empty channel and proceeds to realize the communication by a memory copy. It then reschedules the waiting process and continues its own execution if it is not pre-empted by the rescheduled process.

Alternation is achieved by mimicking the appropriate transputer instructions. An alternation always commences with an alternation start primitive. An alternation guard can be a channel guard, a timer guard or a skip guard, and is also associated with a boolean variable. Each guard of the alternation is enabled in turn and then the alternation waits for a guard to become active. This is followed by a disable sequence for each guard during which the first active guard found is selected. The alternation is terminated by an alt finish instruction which has the effect of branching to the code associated with the selected guard. The enable sequence and the disable sequence are each executed atomically. There is therefore no need to handle the case where a high priority guard becomes active during the disable sequence. Note that this reproduces the bug[4] in the transputer implementation associated with the interaction between `PRI ALT` and `PRI PAR`.

3 Translating Occam to C

This section gives a brief overview of the components of the occam compilation system. Figure 1 illustrates the coupling between the various modules from which the compilation system is constructed.

FRONT END	Phase 1	Lexer	Hand-coded
	Phase 2	Parser	`lalr` & `puma` & `ast`
	Phase 3	Type Checker	`ag` & `puma`
	Phase 4	Usage Checker	`ag` & `puma`
BACK END	Phase 5	occam-occam Translation	`puma`
	Phase 6	C Attribution	`ag`
	Phase 7	C Generation	`puma`

Figure 1: Compiler structure

3.1 Lexical Analysis

Lexical analysis is performed by a hand-coded token generator. In addition to tokenisation of the source file, the lexer performs a number of additional tasks, such as conversion of string and numeric constants into binary format. Indentation is converted into tokens for `BEGIN` and `END` of scopes; a record of indentation levels is kept in order to support nested `VALOF`s.

A number of compiler directives are understood by the lexical analyser, including `#INCLUDE`, which causes textual inclusion of a file, `#USE`, which pulls in and tokenises a prototype file. The `#C` and `#H` directives pass strings directly through to the code-generator where a simple form of parameter substitution is performed, to allow direct inclusion of `C` language constructs.

Several parser ambiguities and special-cases are corrected within the lexical analyser through the use of lexical look-ahead.

3.2 Parsing and Syntax Tree Generation

The parser, generated from a high-level `BNF`-like description with corresponding hand-coded semantic actions, constructs an abstract syntax tree using the tree construction primitives made available by the `ast` tool. A symbol table, using occam scoping rules is also generated during this process, and symbol references are resolved into symbol table indices.

The occam grammer is not strictly *LL(1)* and several grammatical ambiguities have to be resolved by a combination of lexical lookahead and syntax tree patching. The following code fragments show two of the more serious ambiguities.

```
...
CASE
 INT a,b :
 INT d,e
 SEQ
 ...
```

```
...
CASE
 [2][3]INT a :
 [1][0]
 SEQ
 ...
```

In the first case, a naive LL(1) parser cannot differentiate between the variable declaration and the `INT` conversion, causing a grammer conflict. This can be resolved by the lexer recognising the terminating colon on the declaration line and feeding a modified token stream to the parser.

The second example cannot be resolved by the lexer. In this case, the syntax tree representation of the square-bracketted expression depends upon the parsing context and the parser is unable to diffentiate the two lines until it has already parsed and constructed a syntax tree branch representing the expressions. This requires the syntax tree to be patched once the parser is able to differentiate the two statements, converting a list of array bounds into a single element constant table with a constant subscript.

3.3 Tree Attribution, Checking and 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. Checks are performed where possible on the validity of these attributes.

3.3.1 Type Checking

The type-checking stage of the compiler propagates various attributes around the syntax tree. Synthesized type information is generated and checked to detect any type violations in the code. Element writability is synthesized and checked in a similar way. The direction of channel usage is checked to ensure that each channel has, at most, one reader and one writer.

3.3.2 Usage Checking

Usage and alias checking of both scalars and arrays is implemented by propagation of usage information around the abstract syntax tree. Symbolic checking of scalar variables is complete and accomplished by the propagation of two usage attributes around the tree, the set of variables read within any piece of sequential code and a corresponding set of variables written. The propagation is done by union and intersection operations on these sets performed when required. Checking of array usage is currently restricted to simple expressions involving constants and linear functions of replicator indices, and is accomplished by propagating lists of array access slices.

Symbolic and Array alias checking of parameters against non-locals and parameters against other parameters is performed. This handles both explicit and implicit aliasing (`IS` Abbreviations).

Linear replicator expressions are converted to a standard form in preparation for parallel usage checking and allowing range checking to be done both for subscripts of linear replicators and for constant subscript expressions.
e.g. `((2*k)+(((i-j)*5)+4)` $\Rightarrow$ `4+5.i-5.j+2.k`

3.3.3 *Symbolic Parallel Usage Checking*

The following pseudo `ag` code performs parallel usage checking, between processes in a `PAR`.

```
DECL ProcList = [W,R:tSet INHERITED].
RULE
Par = { ProcList:W_in:=∅; ProcList:R_in:=∅; }.

AProc =
{
  CHECK (W_this ∩ W_in ≠ ∅) => write/write error;
  CHECK (W_this ∩ R_in ≠ ∅) ∨ (R_this ∩ W_in ≠ ∅) => read/write error;
  Next:W_in:=W_this ∪ W_in ;
  Next:R_in:=R_this ∪ R_in ;
}.
```

A `Par` node contains a `ProcList` child, which is the list of processes within the `Par`. Each entry in the list consists of an `AProc` node, which inherits usage sets Win,Rin from the previous node in the list and contains local usage sets for the sequential process W_{this},R_{this}. The grammer rules for `AProc` first check for usage errors and then propagate the usage sets to the next `AProc` in the list. The following example shows how this propagation detects Write/Read conflicts and Write/Write conflicts.

	Thread Usage	Propagated Usage	Errors
`INT a,b,c:`			
`PAR`		$W_{in} = \emptyset$,$R_{in} = \emptyset$	
`SEQ` `a:=1`	$W_{this} = \{a\}$,$R_{this} = \emptyset$		
		$W_{in} = \{a\}$,$R_{in} = \emptyset$	
`SEQ` `b:=2`	$W_{this} = \{b\}$,$R_{this} = \emptyset$		
		$W_{in} = \{a, b\}$,$R_{in} = \emptyset$	
`SEQ` `c:=b`	$W_{this} = \{c\}$,$R_{this} = \{b\}$		
		$W_{in} = \{a, b, c\}$,$R_{in} = \{b\}$	W/R
`SEQ` `a:=b`	$W_{this} = \{a\}$,$R_{this} = \{b\}$		
		$W_{out} = \{a, b, c\}$,$R_{out} = \{b\}$	W/W

3.3.4 *The Omega Test*

The Omega Test [11], tests for integer solutions to arbitrary sets of linear equalities and inequalities. The compiler transforms each usage and range check problem into a set of equalities and a set of inequalities, then applies the Omega Test.

```
[100]INT a :
SEQ i = 0 FOR 10
 SEQ j = 2 FOR 20
 SEQ k = 4 FOR 30
 a[k+(10+((3*j)-i))] := 1
```

$$\Rightarrow \quad \begin{array}{l} \exists \ i,j,k: \\ (0 \leq i < 10)\wedge \\ (2 \leq j < 22)\wedge \\ (4 \leq k < 34)\wedge \\ ((10 - i + 3.j + k < 0)\vee \\ (10 - i + 3.j + k \geq 100)) \end{array}$$

Symbolic parallel usage checking within replicated `Par` is straightforward as W set must be $\emptyset$. Array usage checking is performed by transforming each array access into an integer programming problem introducing additional variables as necessary, as shown in the following example.

```
[100]INT a :
SEQ i = 0 FOR 10
 SEQ j = 2 FOR 20
 PAR k = 4 FOR 30
 a[k+(10+((3*j)-i))] := 1
```

$$\Rightarrow \quad \begin{array}{l} \exists \ i,j,k,k\prime: \\ (0 \leq i < 10)\wedge \\ (2 \leq j < 22)\wedge \\ (4 \leq k < 34) \wedge (4 \leq k\prime < 34)\wedge \\ (k \neq k\prime)\wedge \\ (10 - i + 3.j + k = 10 - i + 3.j + k\prime) \end{array}$$

3.3.5 occam to occam Transformation

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 and strings in expressions are also pushed out into new declarations.

Further transformations are performed in order to simplify the syntax tree where possible. Arbitrary table and string accesses are simplified; in particular, constant table and string subscripts are reduced to their lowest form e.g.
`[[a FROM 20 FOR 30] FROM 5 FOR 10][2]` $\Rightarrow$ `a[(20+5)+2]`

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 manoeuvre the control flow between arbitrary descheduling points 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[4]. This phase also defines the contents of

[4]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.

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. Atomic-procedure and function instances pass any parameters using the C parameter mechanism, whereas non-atomic procedure 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.

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 2. Additionally, the occam compiler can be recompiled to map `INT64`s into `INT32`s if the C compiler does not support the `long long` type.

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 2: 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.

3.6 C Compilation, Linking and Execution

Compilation of the ANSI C source files generated by the translator should be straightforward. 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. A utility program `omakef` parses occam source hierarchies for any inter-module or library dependencies and generates a `makefile` to build an executable or library.

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.

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 with debugging enabled then the debugging techniques described in the user-guide [3] can be employed.

4 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.

4.1 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 3.

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

Figure 3: Performance for highly concurrent example.

4.2 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 4 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 4: Performance for network simulation.

4.3 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[2]. 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 5 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 5: Performance for a CFD simulation.

5 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.

6 Acknowledgements

This work has been supported by the European Commission Esprit Project *GPMIMD* P5404.

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

William Pugh et al. at the Department of Computer Science, **University of Maryland**, College Park for the Omega test.

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

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.

The lexical analyser is loosely based on that developed by Peter Polkinghorne.

References

[1] Barrett, G. *Formal methods applied to a floating point number system: Technical Monograph*. PhD thesis, Oxford University Computing PRG, 1987.

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

[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 GPMIMD:013/O2C/GUIDE Revision 0.1, University Of Southampton, Oct. 1993.

[4] G. Barrett, M. Goldsmith, G. J., and Kay, A. The meaning and implementation of pri alt in occam. In *Proceedings of the 9th occam user group technical meeting* (1988), pp. 37–46.

[5] Grosch, J., and Emmelmann, H. *A Tool Box for Compiler Construction*, vol. 477. Springer, Oct. 1990, pp. 106–116.

[6] Hagersten, E., Landin, A., and Haridi, S. DDM - a cache-only memory architecture. *IEEE Comput. Mag. 25*, 9 (Sept. 1992), 44–54.

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

[8] Johnson, S. C. Yacc - Yet another compiler compiler. Computing science technical report no. 32, AT&T Bell Laboratories, 1975.

[9] Lesk, M. E. Lex - A lexical analyzer generator. Computing science technical report no. 39, AT&T Bell Laboratories, 1975.

[10] Meiko. CS-2 product description. Bristol, United Kingdom, 1992.

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

[12] SunSoft. *SunOS 5.2 Reference Manual*.