Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006
© 2006 The authors. All rights reserved.

Native Code Generation Using the Transterpreter

Christian L. JACOBSEN, Damian J. DIMMICH and Matthew C. JADUD

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NZ, England.
{clj3, djd20, mcj4} @kent.ac.uk

Abstract. We are interested in languages that provide powerful abstractions for concurrency and parallelism that execute everywhere, efficiently. Currently, the existing runtime environments for the occam-π programming language provide either one of these features (portability) or some semblance of the other (performance). We believe that both can be achieved through the careful generation of C from occam-π, and demonstrate that this is possible using the Transterpreter, a portable interpreter for occam-π, as our starting point.

Keywords. Transterpreter, native code, GCC, occam-π

Introduction

The Transterpreter [1] is a virtual machine for CSP [2] and Pi-calculus [3] based languages. Intended as a platform for concurrent language design and research, it also fully supports the execution of programs written in the occam-π [4] language. To support our research goals, the Transterpreter was designed to be extensible and portable. This does mean that it is unlikely to be able to achieve the level of performance that could be obtained by compiling directly to native code. However, what the Transterpreter lacks in performance, it possesses in portability, and has enabled the execution of occam-π on platforms as diverse as the Mac OS X (PowerPC) and the LEGO Mindstorms (H8/300) [5]. Porting the Transterpreter to a POSIX compliant OS, such as Mac OS X, is no harder than recompiling 4000 lines of ANSI C. Where the platform is more specialised, such as the LEGO Mindstorms, a new wrapper must be written to interface with the platform appropriately. Such a wrapper can be as short as 100 lines of code.

Techniques to improve the performance of the Transterpreter are being constantly evaluated. Such performance optimisations are however only implemented where they do not sacrifice portability. As an example, Just-In-Time (JIT) compilation may not be practical on all platforms, and must therefore be implemented in a way that does not make the virtual machine unportable. Additionally, recent ports of the Transterpreter to the Tmote Sky [6], a wireless sensor network platform based on the Texas Instruments MSP430 [7], and to the multi-core Cell Broadband Engine [8,9], have identified platforms with specialised needs on which interpretation provides a suboptimal solution. On small battery powered devices, power conservation is likely to be a major concern and the overhead of interpretation may therefore not be acceptable. The Cell Broadband Engine, on the other hand, contains specialised vector processing units, capable of performing high performance vector operations on 128-bit wide data. These units are also capable of running scalar code, and therefore the Transterpreter, but they do so with a large performance penalty. In order to effectively use occam-π on the Cell Broadband Engine, on wireless sensor networks and in a host of other specialised applications, it seems necessary to be able to generate efficient native code.

Given the diverse run-time environments we wish to target, the goal becomes not just the generation of native code, but to realise a code-generation framework which can, without undue effort, target a wide variety of architectures. Our ultimate goal is therefore to create a native code compiler which is as portable as the Transterpreter, and yet generates executables with performance comparable to the existing native code compilers. In this paper we will therefore explore the steps necessary for generating efficient native code, in a retargetable manner.

1. An Overview of occam Environments

A number of occam compilers and runtimes exist which allow occam programs to be interpreted or executed natively. These environments vary in terms of their portability and performance, as well as the assistance they can offer the developer in implementing sound concurrency. Each environment presents its own challenges as a starting point for developing an efficient, portable code generator for occam inspired languages.

1.1. Existing Compilers

The most prominent occam compiler currently available is the "Kent Retargetable occam-π Compiler", KRoC [10,11], which is being developed at the University of Kent, originally as part of the occam-for-all project [12]. KRoC compiles both occam2.1 and the new dynamic occam-π language [4] to native code using the custom `tranx86` [13] code generator. `tranx86` generates code for the IA-32 architecture; code-generators for PowerPC, MIPS and SPARC targets are in development. SPOC, the "Southampton Portable occam Compiler" [14,15], implements the occam2 and occam2.1 languages [16] by compiling them to ANSI C code. This allows SPOC to generate code for a wide variety of platforms, even though it has not been under active development for some time.

Other occam compilers exist, notably the occam1 [17] compiler in the Amsterdam Compiler Kit [18,19,20], which can generate native code for the IA-32 and a number of more archaic architectures. occam compilers which do not generate code for modern architectures are mostly of historical interest, and include the original Inmos compilers, written in occam, and later rewritten in C. occam has also found use in compiler courses, as is the case with the μoccam dialect developed and used at the University of Edinburgh [21].

A number of new occam-inspired compilers are currently being developed. **42** [22], is a small extensible compiler aimed at concurrent language design and development. It implements a small CSP-based language as the foundations for future research. **42** uses the Transterpreter as its underlying runtime, and has supported the work presented in this paper, as the host for the C code-generator. KRoC has approached the point where it is becoming extremely difficult to extend sensibly and the NoCC compiler [23] is being developed at the University of Kent to replace it. Additionally development of the Grid-occam compiler has been reported in [24], as an occam compiler for distributed computing on the .NET platform.

1.2. Towards Native Code

In order to develop a portable and efficient code generator for occam-π programs we have explored the most desirable features of the existing environments described above. This has made it possible to provide an approach which combines the best features from each of the existing implementations.

The virtual machine solutions are generally the most portable, and are often implemented in languages which compile on a wide variety of platforms. Virtual machines, where they have been designed to be cleanly separable functional units, can be viewed as a set of interrelated components for implementing and executing a language. The Transterpreter's scheduler is written entirely in ANSI C, as a set of reusable functions. These scheduler functions will be utilised unmodified in our native code generation approach. As we are investigating fast and portable native code generation, we will not require any other direct support from the Transterpreter.

SPOC compiles to ANSI C code, which is further translated to native code by GCC or some other C compiler. This achieves good performance, especially for sequential code, and provides good portability. The performance of scheduling code is affected by SPOC having to work within the constraints of the C programming language, and it cannot provide the same performance as KRoC. KRoC, and its successor NoCC, make use of custom native code generators capable of generating very efficient scheduler code. While they do optimise sequential code, our testing indicates that they do not achieve the same performance as SPOC, which has most of GCC's optimisations at its disposal. Our intent is to attempt to combine the performance of KRoC's scheduling code with the sequential performance and portability of SPOC.

To generate efficient native code, we look to the approach used in the Amsterdam Compiler Kit, which eliminates the need to implement native code generators for each new language front-end. This is done by using a common intermediary language, which has a persistent form that a language front-end can store to disk. Back-ends can then be invoked on the (architecture independent) intermediary code, in order to generate native code. Thus, effort invested in the code generator will automatically benefit all language front-ends [18]. Unfortunately GCC does not allow language front-ends to be decoupled from back-ends in this manner, as its intermediary language can only be expressed in-memory. Other compiler frameworks provide intermediary formats which are easier to target than GCC, but they generally do not provide code generators for nearly as many platforms.

We would ideally write a new front-end for the GCC compiler, enabling the generation of efficient native occam-π code in a portable manner. This is a large undertaking, and we instead settle for implementing an alternative translation through C. We can then use this to explore the performance that may ultimately be gained from employing the GCC toolchain as our back-end.

2. Using C as an Intermediary Language for occam

C is a useful intermediary language, popular because of its wide availability and portability. Many languages, including the Glasgow Haskell Compiler (GHC) [25], nesC [26] (an event-based language for embedded systems programming) and Mercury [27] (which employs techniques similar to those described in this paper) are compiled first to C, and then to native code using GCC. Using C as an intermediary language can present some specific challenges however, as C was not specifically designed to specifically as an intermediary language. For example, GHC's "Evil Mangler" [28], must modify the assembly instructions generated from C by the GCC compiler, in order to let GHC manage its own stack. In order to translate an occam-π program into C, we must therefore ensure that the C language allows us to fully express the semantics of an occam-π program.

2.1. occam-π Particulars

occam-π uses co-operative scheduling, implemented by well-defined scheduling points in a program. The most common scheduling points are channel communications, which are explicit synchronisation points for occam-π processes. Channel communication is effectively a rendezvous between two processes, where both must have engaged in and completed the communication, before either is allowed to continue. The first process to arrive at a channel communication must therefore block and wait for the process communicating on the other end to arrive. Our occam-π processes will all be multiplexed within a single operating system context, and the execution of the first process must therefore be suspended, and an available process must be scheduled in its stead. This will allow the program to progress to the point where the second process arrives to complete the communication.

An occam-π program must therefore be able to suspend execution of a process at a rescheduling point, and resume execution of some other process at an arbitrary rescheduling point. This is not a natural style of programming to employ in C, which therefore has no explicit support for suspending and resuming execution at well-defined, but arbitrarily paired points in a program.

2.2. Translating to C

When using C as an intermediary language we do not have direct access to the executing program's instruction pointer, and are therefore unable to use it to perform unpredictable jumps between points in a program. We must also honour the C stack, which makes jumping between functions difficult.

ANSI C defines **`goto`** and labels, but these are not particularly useful, as a **`goto`** must reference a named label. This means that a **`goto`** can not jump to an arbitrary location in a program, but only to the specific label it references. Furthermore, **`goto`**s are not allowed to cross a function boundary, as this would most likely invalidate the C stack. The pure ANSI C **`goto`** is therefore not expressive enough to allow the implementation of the transfer of control needed to implement an occam-π scheduler.

While it is possible to use inline assembly constructs to perform arbitrary jumping within a program, the approach taken by CCSP [29], we do not consider this as a feasible solution, as we are extremely attentive to our overall goal of ensuring easy portability. Allowing the use of inline assembly instructions would require that these be re-written for each target architecture. If this assembly becomes non-trivial, we have failed in creating an easily portable native code occam-π compiler.

Use of the `setjmp/longjmp` macros to facilitate transfer of control within a program is also problematic. It is possible to find examples of coroutines implemented using `setjmp/longjmp`. However these rely specific behaviours of the macros, which are not well defined. Furthermore, as noted in [30], the macros are notoriously difficult to implement, and it is unwise to make generalised assumptions about them, making the exotic use of `setjmp/longjmp` likely to be non-portable.

2.3. Using 'switch' Statements

The SPOC compiler has already demonstrated that it is possible to translate occam2 and occam2.1 into ANSI C, and compile it using GCC or some other ANSI C compiler. SPOC will translate occam constructs directly to C where possible, so that occam **`SEQ`**s and **`WHILE`**s are translated into **`for`** or **`while`** loops. SPOC must also deal with the scheduling of occam processes and does this by making heavy use of the C **`switch`** statement. Therefore, the job of the SPOC scheduler is to obtain processes from the scheduling queue or channel queues, and continue its execution by selecting the correct branch of a **`switch`** statement.

The scheduler dispatches execution to a process by first obtaining a pointer to the C function corresponding to it, as well as a pointer to its environment, all wrapped up in the process descriptor taken off the run-queue. The environment contains the `IP` variable, which holds the value of the branch in the function's **`switch`** statement that must be executed next.

When the function is entered (see listing 1), control will be passed to the appropriate branch (**case**) in the **switch** statement. As each branch in the **switch** statement contains one atomic sequence of occam code, control does not need to be transferred until execution completes the statement immediately before the next branch (the `OUTPUT1` macro in listing 1). The arguments to the `OUTPUT1` macro contain the 'label' of the next branch to execute, in this case '1'. The arguments also specify the channel used for communication and the memory location containing the item being communicated. When the `OUTPUT1` macro needs to schedule another process, a C **return** statement is executed, returning control to the scheduler, which is then able to select the next process from the run-queue and resume its execution. The process shown in listing 1 will eventually be placed back on the run-queue, and subsequently executed by the scheduler. When this happens, execution will resume from `case 1`.

```
switch(Env->IP) {
  case 0:
    Env->Temp = 97;
    OUTPUT1(FP->Chan51, &Env->Temp, 1);
  case 1:
    ... /* More cases may follow */
}
```

Listing 1. Part of a simple SPOC process

2.4. *Labels as Values and Computed 'goto's*

Another approach which enables transfer of control between arbitrary scheduling points is to use the GCC 'labels as values' extension. The extension is primarily provided by GCC in order to facilitate the writing of threaded[1] interpreters. Threaded interpreters do away with the traditional fetch-execute loop, and instead 'compile' bytecode into an array of addresses corresponding to the start of an instruction. At the end of the C code implementing a particular instruction, a jump to the next instruction is performed by using a 'computed **goto**': `goto **(pc++);` or similar. This can improve the performance of an interpreter over a pure **switch** based one, as the fetch-execute loop is eliminated. Execution speeds can be further improved weaving the instruction fetch into the code for each instruction, to help the C optimiser fill load and branch delay slots [31,32].

The remainder of this paper will discuss a method for generating C code from occam-π sources using 'labels as values' and 'computed **goto**'.

3. Using the Transterpreter as a Runtime for Native C

This section describes the mechanisms used to transform an occam-π program into a C program that can be compiled into native code by a compiler that supports 'labels as values' and 'computed **goto**'. Both the C compiler found in the GNU Compiler Collection (`gcc`), and the Intel Compiler (`icc`) provide these extensions. Unmodified scheduler functions from the Transterpreter are used to provide the scheduling functionality required for the generated C programs. Because the scheduler functionality is provided as a set of C functions, the C stack must be left intact by the occam-π program. It is therefore not possible to jump across function boundaries using **goto**, and the entire occam-π program must be compiled as a single C function.

The translations described in this paper are used by the **42** compiler to turn occam-π-like programs into C. It would also be possible to translate the Extended Transputer assembly

[1]threading in this context does not relate to multiple execution contexts, i.e. threads, but to a dispatch mechanism used in interpreters.

Code (ETC) [33] generated by the `occ21` compiler in this manner, although such a translator has not been implemented at the time of writing. A translation from ETC is likely to produce poorer quality code, as a great deal of useful semantic information is lost in the translation from occam-π to Extended Transputer assembly Code.

3.1. An Overview of the Execution Environment

The scheduler is implemented as a number of C functions, called using the C stack, which must be available for this purpose. A separate stack (the workspace) is allocated for the occam-π program, indexed using the workspace-pointer (`wptr`). In addition to the workspace-pointer, a number of pointers are needed to hold the scheduling queues. The `fptr` and `bptr`, front- and back-pointer respectively, are used to keep the linked list of runnable processes. Runnable processes are inserted onto the back-pointer, and the next scheduled process is taken off the front-pointer. A timer queue is kept in the `tptr`, with the next timeout stored in `tnext`. The timer queue contains a list of processes ordered by timeout, and care must be taken to insert processes into the correct place in the queue.

Listing 2 shows the state described above as defined in the generated C file. Depending on the architecture, it is possible to place one or more of these pointers in machine registers; shown here is the code generated for an Intel IA-32 compatible processor, where the workspace pointer `wptr` has been placed in the base pointer register `ebp`. On architectures with larger register sets, it is possible to keep a larger amount of scheduling pointers in registers, thereby improving scheduling performance slightly.

```
register int *wptr asm("ebp");
int *fptr, *bptr, *tptr, tnext;
```

Listing 2. Required state

The workspace grows upwards, and the workspace pointer is therefore initialised to point into the bottom of the allocated workspace. The scheduling pointers are initialised to be `null`.

3.2. Simple 'goto's

ANSI C's **`goto`** statement is used to transfer execution between well known points in the program. This is used to implement **`PROC`** invocations and complex loops which we cannot translate into **`while`** or **`for`** loops easily.

3.3. Manipulating State

The program's state, other than that shown in listing 2, is stored in the workspace, and indexed by the `wptr` variable. Storing a value into the workspace becomes a simple assignment using the workspace pointer as an index: `wptr[1] = 2`. Complex expressions are translated directly into C wherever practical, in order to let the C compiler perform any optimisations possible: `wptr[2] = wptr[1] + (wptr[3] / 2);`.

Workspace is allocated and deallocated by modifying the address held in `wptr`. For example, allocating eight slots, each the size of the machine word, is done by subtracting from the `wptr`: `wptr = wptr − 8;`, deallocation by adding: `wptr = wptr + 8;`.

3.4. Manipulating Processes: Labels as Values

The "labels as values" extension provides the ability to take the address of a label by using the `&&` operator. The application of this operator produces the address of a variable, and can be used in arbitrary expressions and assignments. It is therefore possible to take the address of a label which may be needed in the future: `&&L10`.

Once we have obtained such a value, we can use it in, for example, the `add_to_queue(int *wptr, void *iptr);` function, which takes a process descriptor (actually the workspace pointer) of a process, as well as its initial instruction pointer and stores it onto the scheduling queue. This function is used to make a function runnable, by adding it onto the scheduling queue: `add_to_queue((int)(wptr − 4), (int) &&L13);`.

3.5. Calls and Returns: Computed 'goto's

It is often necessary to use the address of a location that appears after the current statement. We can get the address of such locations by dropping labels in the code, which we can then reference using the 'labels as values' extension. Listing 3 shows the code for a call, used to invoke a **PROC**. This code needs to store the return address, which is done by placing the `cur_ip_42` label after the code implementing the call. The call code can then reference this label in order to put it in the return address slot in the new **PROC**s stack frame: `wptr[−4] = (int) &&cur_ip_42;`. The remaining two assignments into the new stack frame are used to pass values to the new **PROC**. Finally the stack frame is allocated by moving the `wptr` up by four slots, and the **goto** is executed.

```
  wptr[−4] = (int) &&cur_ip_42;
  wptr[−3] = wptr[2];
  wptr[−2] = wptr[3];
  wptr = wptr − 4;
  goto L8;
cur_ip_42:
```

Listing 3. Calling a **PROC**

```
wptr = wptr + 4;
goto *wptr[−4];
```

Listing 4. Returning from a **PROC**

The return address now resides in the stack frame allocated by the previous call. Eventually the call will return, at which point the address will be loaded out of the stack and used as an argument to a **goto** (listing 4). A special syntax is used to indicate that the argument of the **goto** is not a label in itself, but rather an expression containing the address of a label, a 'computed **goto**': `goto *expression;`. To return from a call the stack frame is deallocated: `wptr = wptr + 4;`, and the return address is read out from the stack and used in the **goto**.

3.6. Scheduling

'Labels as values' and the 'computed **goto**' are used to deal with scheduling points. An occam-π program has a fixed number of scheduling points, most often in the form of channel communications. When a scheduling point is reached, the location of the currently executing context must be stored so we can return to it later. A previously executing context must then be loaded, so that execution can be resumed. The prototype for the `in` function, which performs this task, is shown in listing 5. This is an explicit scheduling point, where execution will have to transfer elsewhere in the program.

```
char *in(int nbytes, int *chan_ptr, char *write_start, char *iptr);
```

Listing 5. Prototype for the '`in`'put part of a communication

The `in` function takes as arguments, the number of bytes which are being communicated, a pointer to the channel word on which the communication is occurring, a pointer to the location where the data should be put, and finally a pointer to the location where the current process would like execution resumed once the communication has successfully completed. The function returns the address where execution should resume once the `in` has completed successfully. The address returned may be that of the process which engaged the other side of the communication. If this process is not yet ready, an address of an arbitrary process from the scheduling queue will be returned instead. A typical communication is shown in listing 6.

```
 goto *(in(4, (int *) wptr[1], (char *)(wptr + 3), &&id_in_aa));
id_in_aa:
```

Listing 6. Performing a communication

When execution reaches the statement in listing 6, the `in` function will be called. The arguments given here are: `4`, the number of bytes to transfer; `wptr[1]`, dereference of a workspace location, containing the address of the channel being communicated on; `wptr + 3`, the location where the value which will be received is going to be stored; and lastly `&&id_in_aa` the address of the label located immediately following the `in` function. This address is stored by the `in` function, so that when the channel communication has completed and this process is rescheduled, execution can resume from that point, i.e. the label `id_in_aa`. Once in progress, the `in` function will, if the other side of the communication is already waiting, perform the copy of data into `wptr + 3`, and return, using the other sides resume address as the return value. If the other side is not yet ready, the `in` function will park the current process in the channel, and pick off a new process from the run-queue and return its address. It is only at some later point in the execution of the program, when another scheduling function returns the address of the label `id_in_aa` that execution will resume at this point.

3.7. Limitations of This Approach

The approach for native code generation presented in this paper has some limitations. There are a number of ways in which these may be addressed, including the use of the method employed to perform **goto**s across function boundaries, as described in [27]. This approach is not entirely portable however, and in this paper we also discuss the possibility of interfacing directly with existing compiler frameworks (see section 5.1 on the facing page).

While the scope of the approach described in this paper is limited to small monolithic programs, as described below, this does not leave it without merit. We are currently using it give us confidence that a compiler framework such as GCC can provide excellent performance for CSP based languages without an unnecessarily large development overhead. It is also possible to use the method described in this paper to target fairly small devices, such as sensor networks and the individual vector processing units on the Cell Broadband Engine. These platforms can be used for prototyping work, on which we can evaluate the benefit of native code compilation on small, constrained devices, before taking our approach further.

- **Code size limitations**: occam-π programs are being compiled into one large monolithic function in order to enable the use of ‘labels as values’ and ’computed **goto**s’. This can present a problem for GCC, as it compiles and optimises a function at a time, and does not perform well when single functions become very large. Both compilation time, and memory usage can be substantial when compiling large monolithic functions.
- **External linkage**: This is related to the compilation of occam-π programs as one monolithic function. It is not possible to cross a function boundary using a **goto**, and it is therefore not possible to combine several pre-compiled object files together to form a complete native occam-π program. Separate compilation can reduce compilation times, and allow for dynamic (runtime) linkage. Dynamic linkage is not a great concern on the targets we are currently dealing with.
- **Floating point arithmetic**: C does not provide full control over the floating point unit. For example, it is not possible for a program to set rounding modes, or receive

floating point traps. This is potentially problematic for the generation of floating point code from occam-π programs, as occam-π does provide fine grained control over floating point operations.

4. Performance

This section outlines some preliminary performance figures for native code generated using the **42** compiler. We present two benchmarks: commstime and matrix multiply. Commstime is used to measure the context switch time of a CSP based runtime, and highlights the time it takes to switch from one running process to another. We have also included a benchmark which performs a naive matrix multiplication, in order to assess the effectiveness of GCC in optimising generated looping code. The matrix multiply benchmark contains three nested loops which performs a matrix multiplication on two 500 by 500 arrays of integers. The initialisation of the two arrays is included in the measured time. Further benchmarks will be produced as the **42** compiler matures.

The benchmarks have been run against KRoC 1.4.1-pre4, SPOC 1.3, and the Transterpreter 0.7. The benchmarks were executed on an unloaded Intel Pentium 4 CPU running at 3.60GHz, using a Linux 2.6.15 (preemptible) kernel. The first two entries ('Generated C') in tables 1 and 2 refer to the programs compiled to native C using the approach presented in this paper.

Commstime	**nanoseconds**
Generated C, `-O0`	110
Generated C, `-O2`[2]	8
KRoC	14
KRoC, `-io -is -it`	12
SPOC, `-O0`	65
SPOC, `-O2`	31
Transterpreter	147

Table 1. Commstime, context switch times

Matrix Multiply	**milliseconds**
Generated C, `-O0`	1707
Generated C, `-O2`[2]	1052
KRoC	1939
KRoC, `-io -is -it`	1930
SPOC, `-O0`	1486
SPOC, `-O2`	683
Transterpreter	35870
ANSI C `-O0`	1444
ANSI C `-O2`	672

Table 2. Matrix multiplication times

While the benchmark presented are preliminary and should not be taken as conclusive, they do offer an indication that our approach is worth further investigation. The commstime benchmark currently indicates that the approach presented has a context switch time which is slightly faster than KRoC's. The Transterpreter scheduler, which is also used in the generated C code, does not however deal with priority, which may account for some of the difference in speed. The matrix multiply benchmark illustrates the value of relying on optimisations already present in the GCC compiler. This leaves the C generator with the significantly simpler task of producing optimisable code, rather than optimised code.

5. Future Directions

5.1. Using a Compiler Framework

In this paper, we have concentrated on producing fast and portable native code for the occam-π language, and languages like it. We have placed particular emphasis on ensuring portability,

[2]additionally the `--fomit-framepointer` and `--inline-functions` flags have been used.

to enable code-generation for a large number of architectures with minimum effort. In our effort to compile to as many architectures as possible, many existing compiler frameworks become unavailable to us, as they only target four to five of the 'major' architectures (i.e. platforms such as the IA-32, PowerPC and Sparc). None of the compiler frameworks which the authors have investigated, including: `C--` [34], LLVM [35], Zephyr [36] and MLRISC [37], provide the desired level of portability, as compared with the GNU Compiler Collection. They do however, present more friendly compiler targets than GCC.

5.2. Using GCC as a Compiler Framework

The GNU Compiler Collection [38] (GCC) contains a number of language front-ends and target back-ends. GCC currently supports the compilation of C, C++, Java, Ada, ObjectiveC and Fortran95, and generates code for a large number of architectures, listed on the GCC compilers backend status page [39]. Further, a number of front- and back-ends are in development, many with the aim of being integrated into GCC proper, once a suitable point of maturity has been reached. These include front-ends for BCPL [40], Modula-2 [41] and the logic programming language Mercury [42]. Additionally the Treelang front-end serves as an example of how to write a front-end for GCC. Back-ends for the MSP430 [43] and Cell Broadband Engine [44] are under active development.

GCC started out as a C compiler, targeting only the major architectures used by the GNU project, but has in recent years been progressing towards becoming a more generic compiler construction framework. GCC has over the years, in no small part due to its open nature, evolved into a multi-language compiler, which targets over 30 platforms. While it has been notoriously difficult to write a new language front-end for the GCC compiler in the past, better documentation [45,46], example front-ends, and an internal representation less biased towards C/C++-like procedural languages has made this task easier. Writing a language front-end for GCC is still no trivial undertaking, as a language front-end must interface directly with GCC, and produce in memory GENERIC datastructures [47], which can then be lowered to other internal datastructures automatically, in order to perform optimisations and code generation.

5.2.1. Interfacing occam-π with GCC

This paper has been describing a method of interfacing with GCC through the use of C as an intermediary language. This comes at the cost of expressiveness, due to the constraints of the C language. The harder, but more flexible way to use GCC is to interface with it directly, and make the front-end generate GCC's internal datastructures. This however requires an intimate understanding of the internals of GCC, which requires considerably more effort than generating C code. Furthermore, it is yet to be determined if GCC's internal program representation GENERIC is able to represent occam-π programs. Since GENERIC is designed as a language independent representation, this is not likely to be as big an issue as it would have in the past.

We chosen to evaluate the potential performance of using GCC as a back end for occam-π by using C as an intermediary language, rather than starting with the more tricky direct integration into GCC. As we have shown that excellent performance, and portability can be obtained from GCC, we are confident in our ability to provide a front-end for GCC in the future.

6. Conclusions

This paper has demonstrated the feasibility of automatic generation of C code from occam-π programs, which, when compiled to native code, exhibits very good performance, both in terms of context switch times, and sequential code. We have demonstrated that existing compilers and code generators can be used to generate native occam-π programs with performance which compares very favourably with existing solutions.

The potential problems concerning code size limitations and the lack of separate compilation have been identified. In the immediate future however, we are most interested in using this method to target relatively small devices such as the MSP430 and the vector processing units in the Cell Broadband Engine, where these problems are not likely to present a great concern for our prototyping efforts.

Eventually, we would like to overcome these problems by taking a more direct route toward native code than through C, while not sacrificing portability. We will be investigating the use of the currently experimental **42** compiler for the Transterpreter, and making it generate native code using an existing code generation framework. While options such as C-- and LLVM seem like attractive targets, they do not currently target nearly as many languages as the GNU Compiler Collection.

References

[1] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Dr. Ian R. East, Prof David Duce, Dr Mark Green, Jeremy M. R. Martin, and Prof. Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 99–106. IOS Press, Amsterdam, September 2004.

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

[3] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.

[4] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[5] Christian L. Jacobsen and Matthew C. Jadud. Towards concrete concurrency: occam-pi on the lego mindstorms. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 431–435, New York, NY, USA, 2005. ACM Press.

[6] moteiv. Tmote Sky. http://www.moteiv.com/products-tmotesky.php, 2006.

[7] Matthew C. Jadud, Christian L. Jacobsen, and Damian J. Dimmich. Concurrency on and off the sensor network node. *SEUC 2006 workshop*, 2006.

[8] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research & Development*, 49(4/5):589–604, July/September 2005.

[9] Damian J. Dimmich, Christian L. Jacobsen, and Matthew C. Jadud. A cell transterpreter. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, Concurrent Systems Engineering Series. IOS Press, Amsterdam, September 2006.

[10] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166. World occam and Transputer User Group, IOS Press, Netherlands, March 1996. ISBN: 90-5199-261-0.

[11] P.H. Welch and F.R.M. Barnes. The KRoC Home Page, 2006. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`.

[12] Michael D. Poole. Occam for all - Two Approaches to Retargetting the INMOS Compiler. In Brian C. O'Neill, editor, *Proceedings of WoTUG-19: Parallel Processing Developments*, pages 167–178, feb 1996.

[13] Frederick R. M. Barnes. tranx86 – An Optimising ETC to IA32 Translator. In Alan G. Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, pages 265–282, sep 2001.

[14] M. Debbage. Southampton's portable occam compiler (spoc), 1994.

[15] Spoc sourcecode. http://www.hpcc.ecs.soton.ac.uk/software/spoc/.

[16] INMOS Limited. *occam2 Reference Manual*. Prentice Hall, 1984. ISBN: 0-13-629312-3.

[17] INMOS Limited. *occam Programming Manual*. Prentice Hall, 1984.

[18] Andrew S. Tanenbaum, Hans van Staveren, E. G. Keizer, and Johan W. Stevenson. A practical tool kit for making portable compilers. *Commun. ACM*, 26(9):654–660, 1983.

[19] Kees Bot and Edwin Scheffer. An occam compiler. Technical report, Vrije Universiteit Amsterdam, The Netherlands, February 1987.

[20] The Amsterdam Compiler Kit. http://tack.sourceforge.net/, 2006.
[21] Ian Stark and Kevin Mitchell. uOCCAM, CS3 individual programming project, 2000-2001.
[22] Matthew C. Jadud, Damian J. Dimmich, and Christian L. Jacobsen. **42** compiler homepage. http://www.transterpreter.org/wiki/42, 2006.
[23] Fred Barnes. NOCC compiler homepage. http://www.cs.kent.ac.uk/projects/ofa/nocc/, 2006.
[24] Peter Tröger, Martin von Löwis, and Andreas Polze. The grid-occam project. In Mario Jeckle, Ryszard Kowalczyk, and Peter Braun II, editors, *GSEM*, volume 3270 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 2004.
[25] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 93.
[26] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems, 2003.
[27] Fergus Henderson, Thomas Conway, and Zoltan Somogyi. Compiling logic programs to C using GNU C as a portable assembler. In *ILPS'95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming*, pages 1–15, Portland, Or, 1995.
[28] Manuel M. T. Chakravarty, Sigbjorn Finne, Simon Marlow, Simon Peyton Jones, Julian Seward, and Reuben Thomas. The Glasgow Haskell Compiler commentary — the Evil Mangler. http://www.cse.unsw.edu.au/c̃hak/haskell/ghc/comm/the-beast/mangler.html, May 2005.
[29] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press.
[30] Samuel P Harbison and Guy L Steele. *C: A Reference Manual*. Prentice Hall, Upper Saddle River, NJ, USA, fifth edition, February 2002.
[31] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300, New York, NY, USA, 1998. ACM Press.
[32] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.
[33] M.D.Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications*, volume 52 of *Concurrent Systems Engineering*, Address, April 1998. WoTUG, IOS Press.
[34] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C--: A portable assembly language that supports garbage collection. In *PPDP '99: Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, pages 1–28, London, UK, 1999. Springer-Verlag.
[35] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.
[36] A. Appel and J. Davidson. The Zephyr compiler infrastructure. 1999.
[37] Lal George and Allen Leung. MLRISC a framework for retargetable and optimizing compiler back ends. Technical report, January 2003.
[38] GCC Team. GNU compiler collection homepage. http://gcc.gnu.org/, 2006.
[39] GCC Team. GCC backends: Status of supported architectures from maintainers' point of view, April 2006. Available from: http://gcc.gnu.org/backends.html.
[40] Thomas Crick. A BCPL front end for GCC . Bsc (hons) computer science project, University of Bath, May 2004.
[41] Gaius Mulley. A report on the progress of GNU Modula-2 and its potential integration into GCC. In *Proc. GCC Developers' Summit*, pages 109–124, Ottawa, Canada, June 2006. University of Glamorgan.
[42] The Mercury Project. Mercury front-end for GCC. http://www.mercury.cs.mu.oz.au/download/gcc-backend.html, 2006.
[43] Mspgcc homepage. http://mspgcc.sourceforge.net/.
[44] Ulrich Weigand. Porting the GNU tool chain to the Cell architecture. In *Proc. GCC Developers' Summit*, pages 185–189, Ottawa, Canada, June 2005. IBM Deutschland Entwicklung GmbH.
[45] Andreas Bauer. Creating a portable programming language using open source software. *Proceedings of USENIX 2004 Annual Technical Conference*, pages 103–113, June 2004.
[46] Free Software Foundation, Inc. GCC internals. A GNU manual, 2005.
[47] Jason Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *Proc. GCC Developers' Summit*, pages 171–180, Ottawa, Canada, June 2005. Red Hat, Inc.