Translating ETC to LLVM Assembly

Carl G. RITSON

School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, England.

c.g.ritson@kent.ac.uk

Abstract. The LLVM compiler infrastructure project provides a machine independent virtual instruction set, along with tools for its optimisation and compilation to a wide range of machine architectures. Compiler writers can use the LLVM's tools and instruction set to simplify the task of supporting multiple hardware/software platforms. In this paper we present an exploration of translation from stack-based Extended Transputer Code (ETC) to SSA-based LLVM assembly language. This work is intended to be a stepping stone towards direct compilation of occam-π and similar languages to LLVM's instruction set.

Keywords. concurrency, ETC, LLVM, occam-pi, occ21, tranx86

Introduction and Motivation

The original occam language toolchain supported a single processor architecture, that of the INMOS Transputer [1,2]. Following INMOS's decision to end development of the occam language, the sources for the compiler were released to the *occam For All* (oFA) project [3]. The oFA project modified the INMOS compiler (occ21), adding support for processor architectures other than the Transputer, and developed the basis for today's Kent Retargetable occam Compiler (KRoC) [4].

Figure 1 shows the various compilation steps for an occam or occam-π program. The occ21 compiler generates Extended Tranputer Code (ETC) [5], which targets a virtual Transputer processor. Another tool, tranx86 [6], generates a machine object from the ETC for a target architecture. This is in turn linked with the runtime kernel CCSP [7] and other system libraries.

Tools such as tranx86, octran and tranpc [8], have in the past provided support for IA-32, MIPS, PowerPC and Sparc architectures; however, with the progressive development of new features in the occam-π language, only IA-32 support is well maintained at the time of writing. This is a consequence of the development time required to maintain support for a large number of hardware/software architectures. In recent years the Transterpreter Virtual Machine (TVM), which executes linked ETC bytecode directly, has provided an environment for executing occam-π programs on architectures other than IA-32 [9,10]. This has been possible due to the small size of the TVM codebase, and its implementation in architecture independent ANSI C. Portability and maintainability are gained at the sacrifice of execution speed, a program executed in the TVM runs around 100 times slower its equivalent tranx86 generated object code.

In this paper we present a new translation for ETC bytecode, from the virtual Transputer instruction set, to the LLVM virtual instruction set [11,12]. The LLVM compiler infrastructure project provides a machine independent virtual instruction set, along with tools for its optimisation and compilation to a wide range of machine architectures. By targeting a virtual instruction set that has a well developed set of platform backends, we aim to increase

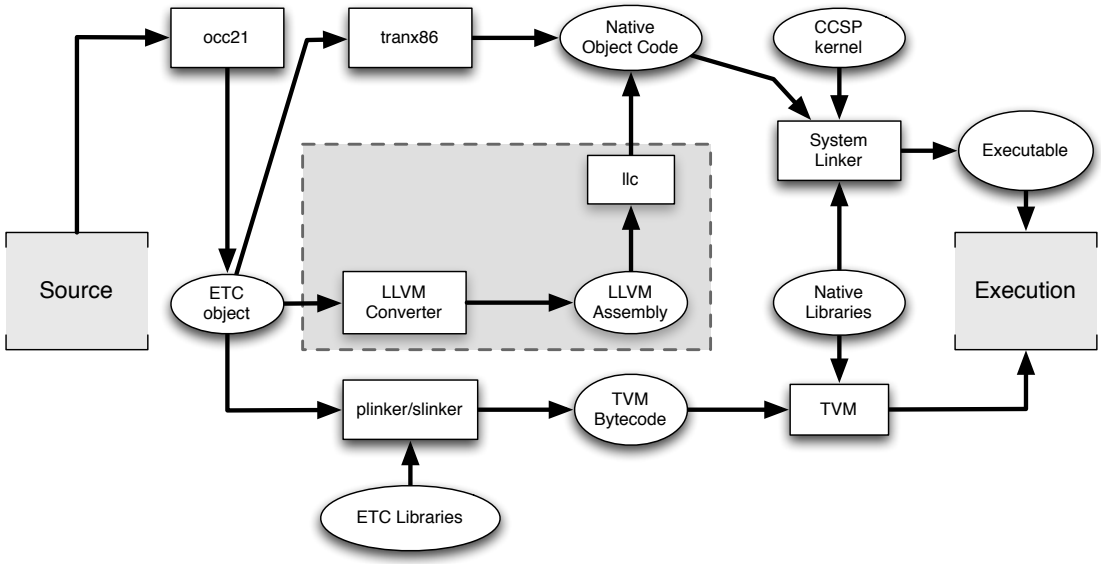

Figure 1. Flow through the KRoC and Transterpreter toolchains, from source to program execution. This paper covers developments in the grey box.

the number of platforms the existing occam-π compiler framework can target. LLVM also provides a pass based framework for optimisation at the assembly level, with a large number of pre-written optimisation passes (e.g. deadcode removal, constant folding, etc). Translating to the LLVM instruction set provides us with access to these ready-made optimisations as opposed to writing our own, as has been done in the past [6].

The virtual instructions sets of the Java Virtual Machine (JVM) or the .NET's Common Language Runtime (CLR) have also been used as portable compilation targets [13,14]. Unlike LLVM these instruction sets rely on a virtual machine implementation and do not provide a clear path for linking with our efficient multicore language runtime [7]. This was a key motivating factor in choosing LLVM over the JVM or CLR.

An additional concern regarding the JVM and CLR is that large parts of their code bases are concerned with language features not relevant for occam-π, e.g. class loading or garbage collection. Given our desire to support small embedded devices (section 3.7), it seems appropriate not to encumber ourselves with a large virtual machine. LLVM's increasing support for embedded architectures, XMOS's XCore processor in particular [15], provided a further motivation to choose it over the JVM or CLR.

In section 1 we briefly outline the LLVM instruction set and toolchain. We describe the steps of our translation from ETC bytecode to LLVM assembly in section 2. Section 3 contains initial benchmark results comparing our translator's output via LLVM to that from tranx86. Finally, in section 4 we conclude and comment on directions for future work.

1. LLVM

In this section we briefly describe the LLVM project's infrastructure and its origins. Additionally, we give an introduction to LLVM assembly language as an aid to understanding the translation examples in section 2.

Lattner proposed the LLVM infrastructure as a means of allowing optimisation of a program not just at compile time, but throughout its lifetime [11]. This includes optimisation at compile time, link time, runtime and offline optimisation. Where offline optimisations may tailor a program for a specific system, or perhaps apply profiling data collected from previous executions.

```
define i32 @cube (i32 %x) {
        %sq = mul i32 %x, %x   ; multiply x by x
        %cu = mul i32 %sq, %x  ; multiply sq by x
        ret i32 %cu            ; return cu
}
```

Figure 2. Example LLVM function which raises a value to the power of three.

The LLVM infrastructure consists of a virtual instruction set, a bytecode format for the instruction set, front-ends which generate bytecode from sources (including assembly), a virtual machine and native code generators for the bytecode. Having compiled a program to LLVM bytecode it is then optimised before being compiled to native object code or JIT compiled in the virtual machine interpreter. Optimisation passes take bytecode (or its in-memory representation) as input, and produce bytecode as output. Each pass may modify the code or simply insert annotations to influence other passes, e.g. usage information.

In this paper we discuss the generation of LLVM assembly language from ETC for use with LLVM's native code generators. The LLVM assembly language is strongly typed, and uses static single-assignment (SSA) form. It has neither machine registers nor an operand stack, rather identifiers are defined when assigned to, and this assignment may occur only once. Identifiers have global or local scope; the scope of an identifier is indicated by its initial character. The example in Figure 1, shows a global function `@cube` which takes a 32-bit integer (given the local identifier `%x`), and returns it raise to the power of three. This example also highlights LLVM's type system, which requires all identifiers and expressions to have explicitly specified types.

LLVM supports the separate declaration and definition of functions: header files declare functions, which have a definition at link time. The use of explicit functions, as opposed to labels and jump instructions, frees the programmer from defining a calling convention. This in turn allows LLVM code to transparently function with the calling conventions of multiple hardware and software ABIs.

In addition to functions LLVM provides a restricted form of traditional labels. It is not possible to derive the address of an LLVM label or assign a label to an identifier. Furthermore the last statement of a labelled block must be a branching instruction, either to another label or a return statement. These restrictions give LLVM optimisations a well-defined view of program control flow, but do present some interesting challenges (see section 2.2).

In our examples we have, where appropriate, commented LLVM syntax; however, for a full definition of the LLVM assembly language we refer the reader to the project's website and reference manual [16].

2. ETC to LLVM Translation

This section describes the key steps in the translation of stack-based Extended Transputer Code (ETC) to the SSA-form LLVM assembly language.

2.1. Stack to SSA

ETC bases its execution model on that of the Transputer, a processor with a three register stack. A small set of instructions have coded operands, but the majority consume (pop) operands from the stack and produce (push) results to it. A separate data stack called the *workspace* provides the source or target for most load and store operations.

Blind translation from a stack machine to a register machine can be achieved by designating a register for each stack position and shuffling data between registers as operands are

```
LDC 0    ; load constant 0
LDL 0    ; load workspace location 0
LDC 64   ; load constant 64
CSUB0    ; assert stack 1 < stack 0, and pop stack
LDLP 3   ; load a pointer to workspace location 3
BSUB     ; subscript stack 0 by stack 1
SB       ; store byte in stack 1 to pointer stack 0
```

Figure 3. Example ETC code which stores a 0 byte to an array. The base of the array is workspace location 3, and offset to be written is stored in workspace location 0.

```
LDC 0    ; () => (reg_1)              STACK = <reg_1>
LDL 0    ; () => (reg_2)              STACK = <reg_2, reg_1>
LDC 64   ; () => (reg_3)              STACK = <reg_3, reg_2, reg_1>
CSUB0    ; (reg_3, reg_2) => (reg_2)  STACK = <reg_2, reg_1>
LDLP 3   ; () => (reg_4)              STACK = <reg_4, reg_2, reg_1>
BSUB     ; (reg_4, reg_2) => (reg_5)  STACK = <reg_5, reg_1>
SB       ; (reg_5, reg_1) => ()       STACK = <>
```

Figure 4. Tracing the stack utilisation of the ETC in Figure 3, generating a register for each unique operand.

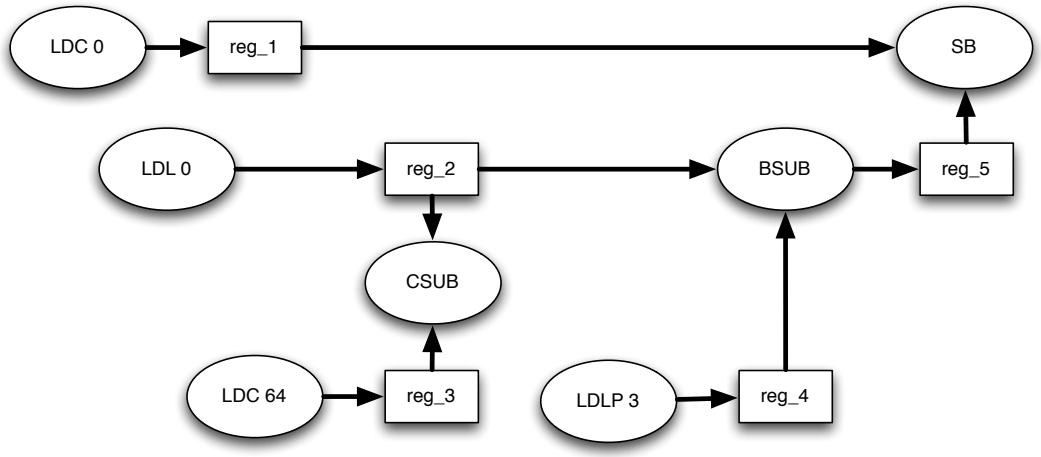

Figure 5. Data flow graph generated from the trace in Figure 4.

.

pushed and popped. The resulting translation is not particularly efficient as it has a large number of register-to-register copies. More importantly, this form of blind translation is not possible with LLVM's assembly language as identifiers (registers) cannot be reassigned. Instead we must trace the stack activity of instructions, creating a new identifier for each operand pushed and associate it with each subsequent pop or reference of that operand. This is possible as all ETC instructions consume and produce constant numbers of operands.

The process of tracing operands demonstrates one important property of SSA, its obviation of data dependencies between instructions. Figures 3, 4 and 5 show respectively: a sample ETC fragment, its traced form and a data flow graph derived from the trace. Each generated identifier is a node in the data flow graph connected to nodes for its producer and consumer nodes. From the example we can see that only the `SB` instruction depends on the first `LDC`, therefore it can be reordered to any point before the `SB`, or in fact constant folded. This direct mapping to the data flow graph representation, is what makes SSA form desirable for pass-based optimisation.

We apply this tracing process to the standard operand stack and the floating point operand

```
; load workspace offset 1
%reg_1 = load i32* (getelementptr i32* %wptr_1, i32 1)
; add 1
%reg_2 = add i32 %reg_1, 1
; store result to workspace offset 2
store i32 %reg_2, (getelementptr i32* %wptr_1, i32 2)

; load workspace offset 3
%reg_3 = load i32* (getelementptr i32* %wptr_1, i32 3)
; synchronise barrier
call void kernel_barrier_sync (%reg_3)

; load workspace offset 1
%reg_4 = load i32* (getelementptr i32* %wptr_1, i32 1)
; add 2
%reg_5 = add i32 %reg_4, 2
; store result to workspace offset 2
store i32 %reg_5, (getelementptr i32* %wptr_1, i32 2)
```

Figure 6. LLVM code example which illustrates the dangers of optimisation across kernel calls.

stack. A data structure in our translator provides the number of input and output operands for each instruction. Additionally, we trace modifications to the workspace register redefining it as required.

Registers from the operand stack are typed as 32-bit integers (`i32`), and operands on the floating point stack as 64-bit double precision floating point numbers (`double`). The workspace pointer is an integer pointer (`i32*`). When an operand is used as memory address it is cast to the appropriate pointer type. In theory, these casts may hinder certain kinds of optimisations, but we have not observed this in practice.

2.2. Process Representation

While the programmer's view of occam is one of all processes executing in parallel, this model is in practice simulated by one or more threads of execution moving through the concurrent processes. The execution flow may leave processes at defined instructions, reentering at the next instruction. The state of the operand stack after these instructions is undefined. Instructions which deschedule the process, such as channel communication or barrier synchronisation, are implemented as calls to the runtime kernel (CCSP) [7]. In a low-level machine code generator such as tranx86, the generator is aware of all registers in use and ensures that their state is not assumed constant across a kernel call. Take the example in Figure 6, there is a risk the code generator may choose to remove the second load of workspace offset 1, and reuse the register from the first load. However the value of this register may have been changed by another process which is scheduled by the kernel before execution returns to the process in the example.

While the system ABI specifies which registers should be preserved by the callee if modified, the kernel does not know which registers will be used by other processes it schedules. If the kernel is to preserve the registers then it must save all volatile registers when switching processes. This requires space to be allocated for each process's registers, something the occam compiler does not do as the instruction it generated was clearly specified as to undefine the operand stack. More importantly, the code to store a process's registers must be rewritten in the system assembly language for each platform to be supported. Given our goal of minimal maintenance portability this is not acceptable.

Our solution is to breakdown monolithic processes into sequences of uninterruptable

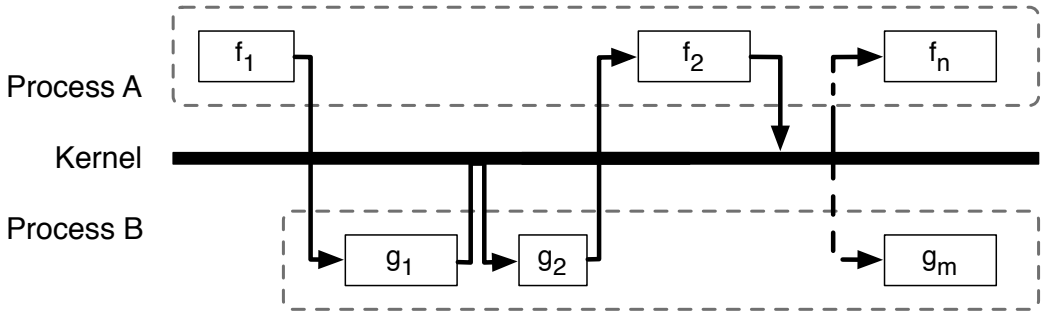

Figure 7. Execution of the component functions of processes A and B is interleaved by the runtime kernel.

functions which pass continuations [17]. Control flow is then restricted such that it may only leave or enter a process at the junctures between its component functions. The functions of the process are then mapped directly to LLVM function definitions, which gives LLVM an identical view of the control flow to that of our internal representation. LLVM's code generation backends will then serialise state at points where control flow may leave the process. Figure 7 gives a graphical representation of this process, as the runtime kernel interleaves the functions f_1 to f_n of process A with g_1 to g_m of process B.

In practice the continuation is the workspace ($wptr$), with the address of the next function to execute stored at $wptr_{[-1]}$. This is very similar to the Transputer's mechanism for managing blocked processes, except the stored address is a function and thus the dispatch mechanism is not a jump, but a call. Thus the dispatch of a continuation ($wptr$) is the tail call: $wptr_{[-1]}(wptr)$.

We implement the dispatch of continuations in the generated LLVM assembly. Kernel calls return the next continuation as selected by the scheduler, which is then dispatched by the caller. This removes the need for system specific assembly instructions in the kernel to modify execution flow, and thus greatly simplifies the kernel implementation. The runtime kernel can then be implemented as a standard system library.

Figure 8 shows the full code listing of a kernel call generated by our translation tool. Two component functions of a process `kroc.screen.process` are shown (see section 2.5.1 for more details on function naming). The first constructs a continuation to the second, then makes a kernel call for channel input and dispatches the returned continuation.

2.3. Calling Conventions

When calling a process as a subroutine, we split the present process function and make a tail call to the callee passing a continuation to newly created function as the return address. This process is essentially the same the Transputer instructions `CALL` and `RET`. There are are however some special cases which we address in the remainder of this section.

The occam language has both processes (`PROC`) and functions (`FUNCTIONS`). Processes may modify their writable (non-`VAL`) parameters, interact with their environment through channels and synchronisation primitives, and go parallel creating concurrent subprocesses. Functions on the other hand may not modify their parameters or perform any potentially blocking operations or go parallel, but may return values (processes do not return values).

While it is possible to implement occam's pure functions in LLVM using the normal call stack, we have not yet done so for pragmatic reasons. Instead we treat function calls as process calls. Function returns are then handled by rewriting the return values into parameters to the continuation function.

The main obstacle to supporting pure functions is that the *occ21* compiler lowers functions to processes, this obscures functions in the resulting ETC output. It also allows some

```
; Component function of process "kroc.screen.process"
define private fastcc void @O_kroc_screen_process_L0.3_0
                           (i8* %sched, i32* %wptr_1) {
  ; ... code omitted ...

  ; Build continuation
  ; tmp_6 = pointer to workspace offset -1
  %tmp_6 = getelementptr i32* %wptr_1, i32 -1
  ; tmp_7 = pointer to continuation function as byte pointer
  %tmp_7 = bitcast void (i8*, i32*)* @O_kroc_screen_process_L0.3_1 to i8*
  ; tmp_8 = tmp_7 cast to an 32-bit integer
  %tmp_8 = ptrtoint i8* %tmp_7 to i32
  ; store tmp_8 (continuation function pointer) to workspace offset -1
  store i32 %tmp_8, i32* %tmp_6

  ; Make kernel call
  ; The call parameters are reg_8, reg_7 and reg_6
  ; The next continuation is return by the call as  tmp_9
  %tmp_9 = call i32* @kernel_Y_in
                     (i8* %sched, i32* %wptr_1,
                      i32 %reg_8, i32 %reg_7, i32 %reg_6)

  ; Dispatch the next continuation
  ; tmp_10 = pointer to continuation offset -1
  %tmp_10 = getelementptr i32* %tmp_9, i32 -1
  ; tmp_12 = pointer to continuation function cast as 32-bit integer
  %tmp_12 = load i32* %tmp_10
  ; tmp_11 = pointer to continuation function
  %tmp_11 = inttoptr i32 %tmp_12 to void (i8*, i32*)*
  ; tail call tmp_11 passing the continuation (tmp_9) as its parameter
  tail call fastcc void %tmp_11 (i8* %sched, i32* %tmp_9) noreturn
  ret void
}

; Next function in the process "kroc.screen.process"
define private fastcc void @O_kroc_screen_process_L0.3_1
                           (i8* %sched, i32* %wptr_1) {
        ; ... code omitted ...
}
```

Figure 8. LLVM code example is actual output from our translation tool showing a kernel call for channel input. This demonstrates continuation formation and dispatch.

kernel operations (e.g. memory allocation) within functions. Hence to provide pure function support, the translation tool must reconstruct functions from processes, verify their purity, and have separate code generation paths for process and functions. We considered such engineering excessive for this initial exploration; however, as the LLVM optimiser is likely to provide more effective inlining and fusion of pure functions we intend to explore it in future work. In particular, the purity verification stage involved in such a translator should also be able to lift processes to functions, further improving code generation.

Another area affected by LLVM translation is the Foreign Function Interface (FFI). FFI allows occam programs to call functions implemented in other languages, such as C [18,19]. This facility is used to access the system libraries for file input and output, networking and graphics. At present the code generator (*tranx86*) must generate not only hardware specific assembly, but structure the call to conform to the operating system specific ABI. LLVM greatly simplifies the FFI call process as it abstracts away any ABI specific logic. Hence

foreign functions are implemented as standard LLVM calls in our translator.

2.4. Branching and Labels

The Transputer instruction set has a relatively small number of control flow instructions:

- `CALL` call subroutine (and a general call variant - `GCALL`),
- `CJ` conditional jump,
- `J` unconditional jump,
- `LEND` loop end (form of `CJ` which uses a counting block in memory),
- `RET` return from subroutine.

In sections 2.2 and 2.3 we addressed the `CALL` and `RET` related elements of our translation, in this section we address the other instructions.

The interesting aspect of the branching instructions `J` and `CJ` are their impact on the operand stack. An unconditional jump undefines the operand stack, this allows a process to be descheduled on certain jumps, which provided a preemption mechanism for long running process on the Transputer. The conditional jump instruction branches if the first element of the operand stack is zero, in doing so it preserves the stack. If it does not branch then it instead pops the first element of the operand stack.

As part of operand tracing during the conversion to SSA-form (see section 2.1), each encountered label within a process is tagged with the present stack operands. For the purposes of tracing, unconditional jumps undefine the stack and conditional jumps consume the entire stack outputting $stackdepth-1$ new operands. Having traced the stack we compare the inputs of each label with inferred inputs from the branch instructions which reference it, adjusting instruction behaviour as required. These adjustments can occur, for example, when the target of a conditional jump does not require the entire operand stack. While the compiler outputs additional stack depth information this is not always sufficient, hence our the introduction of an additional verification stage.

The SSA syntax of LLVM's assembly language adds some complication to branching code. When a label is the target of more than one branching instruction, ϕ nodes (phi nodes) must be introduced for each identifier which is dependent on the control flow. Figure 9 illustrates the use of ϕ nodes in a contrived code snippet generating $1/n$, where the result is 1 when $n = 0$. The ϕ node selects a value for `%fraction` from the appropriate label's namespace, acting as a merge of values in the data flow graph. In our translation tool we use the operand stack information generated for each label to build appropriate ϕ nodes for labels which are branch targets. Unconditional branch instructions are then added to connect these labels together, as LLVM's control flow does not automatically transition between labels.

Transputer bytecode is by design position independent, the arguments passed to start process instructions, loop ends and jumps are offsets from the present instruction. To support these offsets the occam compiler specifies the instruction arguments as label differences, such that $L_t - L_i$ where L_t is the arguments target label and L_i is a label placed before the instruction consuming the jump offset. While we can revert these differences to absolute label reference by removing the subtraction of L_i, LLVM assembly does not permit the derivation of label addresses. This prevents us passing labels as arguments to kernel calls such as start process (`STARTP`). We overcome this by lifting labels, for which the address is required, to function definitions. This is achieved by splitting the process in the same way as is done for kernel calls (see section 2.2). Adjacent labels are then connected by tail calls with the operand stack passed as parameters. There is no need to build continuations for these calls as control flow will not leave the process. Additionally, ϕ nodes are not required as the passing of the operand stack as parameters provides the required renaming.

```
  ; Compare n to 0.0
  %is_zero = fcmp oeq double %n, 0.0
  ; Branch to the correct label
  ;  zero if is_zero = 1, otherwise not_zero
  br i1 %is_zero, label %zero, label %not_zero

zero:
  ; Unconditionally branch to continue label
  br label %continue

not_zero:
  ; Divide 1 by n
  %nz_fraction = fdiv double 1.0, %n
  ; Unconditionally branch to continue label
  br label %continue

continue:
  ; fraction depends on the source label:
  ;   1.0 if the source is zero
  ;   nz_fraction if the source is not_zero
  %fraction = phi double [ 1.0, %zero, %nz_fraction, %not_zero ]
```

Figure 9. Example LLVM code showing the use of a phi node to select the value of the *fraction* identifier.

As an aside, earlier versions of our translation tool lifted all labels to function definition to avoid the complexity of generating ϕ nodes, and avoid tracking the use of labels as arguments. While it appeared that LLVM's optimiser was able to fuse many of these processes back together, it was felt that a layer of control flow was being obscured. In particular this created output which was often hard to debug. Hence, we redesigned our translator to only lift labels when required.

2.5. Odds and Ends

This section contains some brief notes on other interesting areas of our translation tool.

2.5.1. Symbol Naming

While LLVM allows a wide range of characters in symbol names, the generation of symbol names for processes is consistent with that used in *tranx86* [6]. Characters not valid for a C function are converted to underscores, and a `o_` prefix added. This allows ANSI C code to manipulate occam process symbols by name.

Only processes marked as global by the compiler are exported, and internally generated symbols are marked as `private` and tagged with the label name to prevent internal collisions. Declaration `declare` statements are added to the beginning of the assembly output for all processes referenced within the body. These declarations may include processes not defined in the assembly output; however, these will have been validated by the compiler as existing in another ETC source. The resulting output can then be compiled to LLVM bytecode or system assembly and the symbols resolved by the LLVM linker or the system linker as appropriate.

2.5.2. Arithmetic Overflow

An interesting feature of the occam language is that its standard arithmetic operations check for overflow and trigger an error when it is detected. In the ANSI C TVM emulating these arithmetic instructions requires a number of additional logic steps and calculations [10]. This is inefficient on CPU architectures which provide flags for detecting over-

flow. The LLVM assembly language does not provide access to the CPU flags, but instead provides intrinsics for addition, subtraction and multiplication with overflow detection. We have used these intrinsics (`@llvm.sadd.with.overflow`, `@llvm.ssub.with.overflow` and `@llvm.smul.with.overflow`) to efficiently implement the instructions `ADD`, `SUB` and `MUL`.

2.5.3. Floating Point

occam supports a wide range of IEEE floating-point arithmetic and provides the ability to set the rounding mode in number space conversions. While an emulation library exists for this arithmetic, a more efficient hardware implementation was present in later Transputers and we seek to mirror this in our translator. However, we found that LLVM lacks support for setting the rounding mode of the FPU (this is still the case at the end of writing, with LLVM version 2.5). The LLVM assembly language specification defines all the relevant instructions to truncate their results. While not ideal, we use this fact by adding or subtracting 0.5, before converting a value in order to simulate nearest rounding. We do not support *plus* and *minus* rounding modes as the compiler never generates the relevant instructions.

We observed that the *occ21* compiler only ever generates a rounding mode change instruction directly prior to a conversion instruction. Thus instead of generating LLVM code for the mode change instruction we tag the proceeding instruction with the new mode. Hence mode changes become static at the point of translation and can be optimised by LLVM, although this was not done for the purposes of optimisation.

3. Benchmarks

In this section we discuss preliminary benchmark results comparing the output of the existing *tranx86* ETC converter, to the output of our translation tool passed through LLVM's optimiser (*opt*) and native code generator (*llc*). These benchmarks were performed using source code as-is from the KRoC subversion repository revision *6002* [1], with the except of the *mandelbrot* benchmark from which we removed the frame rate limiter.

Table 1 shows the wall-clock execution times of the various benchmarks we will now discuss. All our benchmarks were performed on an eight core Intel Xeon workstation composed of two E5320 quad-core processors running at 1.86GHz. Pairs of cores share 4MiB of L2 cache, giving a total of 16MiB L2 cache across eight cores.

Table 1. Benchmark execution times, comparing tranx86 and LLVM based compilations.

Benchmark	tranx86 (s)	LLVM (s)	Difference (tranx86 → LLVM)
agents 8 32	29.6	27.6	-7%
agents 8 64	91.8	86.5	-6%
fannkuch	1.29	1.33	+3%
fasta	6.78	6.90	+2%
mandelbrot	27.0	8.74	-68%
ring 250000	3.84	4.28	+12%
spectralnorm	23.1	14.3	-38%

3.1. agents

The *agents* benchmark was developed to compare the performance of the CCSP runtime [7] to that of other language runtimes. It is based on the *occoids* simulation developed as part of

[1] `http://projects.cs.kent.ac.uk/projects/kroc/trac/log/kroc/trunk?rev=6002`

the CoSMoS project [20]. A number of agent processes move over a two-dimensional torus avoiding each other, with their behaviour influenced by the agents they encounter. Each agent calculates forces between itself and other agents it can see using only integer arithmetic. The amount of computation increases greatly with the density of agents and hence we ran two variants for comparison. One with 32 initial agents per grid tile on an eight by eight grid, giving 2048 agents, and the other with double the density at 4096 agents on the same size grid. We see a marginal performance improvement in the LLVM version of this benchmark, we attribute this to LLVMs aggressive optimisation of the computation loops.

3.2. fannkuch

The *fannkuch* benchmark is based on a version from *The Computer Language Benchmarks Game* [21,22]. The source code involves a large numbers of reads and writes to relatively small arrays of integers. We notice a very small decrease in performance in the LLVM version of this benchmark. This may be the result of tranx86 generating a more efficient instruction sequence for array bounds checking.

3.3. fasta

The *fasta* benchmark is also taken from *The Computer Language Benchmarks Game*. A set of random DNA sequences is generated and output, this involves array accesses and floating-point arithmetic. Again, like *fannkuch*, we notice a negligible decrease in performance and attribute this to array bounds checks.

3.4. mandelbrot

We modified the occam-π implementation of the mandelbrot set generator in the `ttygames` source directory to remove the frame rate limiter and used this as a benchmark. The implementation farms lines of the mandelbrot set image to 32 worker processes for generation, and buffers allow up to eight frames to be concurrently calculated. The complex number calculations for the mandelbrot set involve large numbers of floating point operations, and this benchmark demonstrates a vast improvement in LLVM's floating-point code generator over tranx86. FPU instructions are generated by tranx86, whereas LLVM generates SSE instructions, the latter appear to be more efficient on modern x86 processors. Additionally, as we track the rounding mode at the source level (see section 2.5.3) we do not need to generate FPU mode change instructions, which may be disrupting FPU pipelining of tranx86 generated code.

3.5. ring

Another CCSP comparison benchmark, this sets up a ring of 256 processes. Ring processes receive a token, increment it, and then forward it on to the next ring node. We time 250,000 iterations of the ring, giving 64,000,000 independent communications. This allows us to calculate the communication times of tranx86 and our LLVM implementation at 60ns and 67ns respectively. We attributed the increase in communication time to the additional instructions required to unwind the stack when returning from kernel calls in our implementation. The tranx86 version of CCSP does not return from kernel calls (it dispatches the next process internally).

3.6. spectralnorm

The final benchmark from *The Computer Language Benchmarks Game*. This benchmark calculates the spectral norm of an infinite matrix. Matrix values are generated using floating-

Table 2. Binary text section sizes, comparing tranx86 and LLVM based compilations.

Benchmark	tranx86 (bytes)	LLVM (bytes)	Difference (tranx86 → LLVM)
agents	16410	36715	+124%
fannkuch	3702	5522	+49%
fasta	5134	10494	+104%
mandelbrot	6098	12865	+111%
ring	3453	6716	+94%
spectralnorm	4065	6318	+55%

point arithmetic by a function which is called from a set of nested loops. The significant performance improvement with LLVM can be attributed to its inlining and more efficient floating-point code generation.

3.7. Code Size

Table 2 shows the size of the text section of the benchmark binaries. We can see that the LLVM output is typically twice the size of the equivalent tranx86 output. It is surprising that this increase in binary size does not adversely affect performance, as it increases the cache pressure. As an experiment we passed a `-code-model=small` option to LLVM's native code generator; however, this made no difference to binary size. Some of the increase in binary size may be attributed to the continuation dispatch code which is inlined within the resulting binary, rather than as part of the runtime kernel as with tranx86. The *fannkuch* and *spectral-norm* benchmarks make almost no kernel calls, therefore contain very few continuation dispatches, and accordingly show the least growth. Another possibility is LLVM aggressively aligning instructions to increase performance. Further investigation is required to establish whether binary size can be reduced, as it is of particular concern for memory constrained embedded devices.

4. Conclusions and Future Work

In this preliminary work we have demonstrated the feasibility of translating the ETC output of the present occam-π compiler into the LLVM project's assembly language. With associated changes to our runtime kernel this work provides a means of compiling occam-π code for platforms other than X86. We see this work as a stepping stone on the path to direct compilation of occam-π using LLVM assembly as part of a new compiler, *Tock* [23]. In particular, we have established viable representations of processes and a kernel calling convention, both fundamental details of any compiled representation of occam-π.

The performance of our translations compares favourably with that of previous work (see section 3). While our kernel call mechanism is approximately 10% slower, loop unrolling enhancements and dramatically improved float-point performance offset this overhead. Typical applications are a mix of communication and computation, which should help preserve this balance. The *occ21* compiler's memory bound model of compilation presents an underlying performance bottleneck to translation based optimisation such as the one presented in this paper. This is a legacy of the Transputer's limited number of stack registers, and it is our intention to overcome this in the new *Tock* compiler. The ultimate aim of our work is to directly compile occam-π to LLVM assembly using Tock, bypassing ETC entirely.

Aside from the portability aspects of this work, access to an LLVM representation of occam-π programs opens the door to exploring concurrency specific optimisations within an established optimisation framework. Interesting optimisations, such as fusing parallel pro-

cesses using vector instructions and removing channel communications in linear pipelines, could be implemented as LLVM passes. LLVM's bytecode has also been used for various forms of static verification, a similar approach many be able to verify aspects of a compiled occam-π program such as the safety of its access to mobile data. Going further, it is likely that LLVM's assembly language may benefit from a representation of concurrency, particularly for providing information to concurrency related optimisations.

Acknowledgements

This work was funded by EPSRC grant EP/D061822/1. We also thank the anonymous reviewers for comments which helped us improve the presentation of this paper.

References

[1] David A. P. Mitchell, Jonathan A. Thompson, Gordon A. Manson, and Graham R. Brookes. *Inside The Transputer*. Blackwell Scientific Publications, Ltd., Oxford, UK, 1990.

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

[3] Michael D. Poole. occam-for-all – Two Approaches to Retargeting the INMOS occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments – Proceedings of WoTUG 19*, pages 167–178, Nottingham-Trent University, UK, March 1996. World occam and Transputer User Group, IOS Press, Netherlands.

[4] Kent Retargetable occam Compiler. (`http://projects.cs.kent.ac.uk/projects/kroc/trac/`).

[5] Michael D. Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In Peter 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.

[6] Frederick R.M. Barnes. tranx86 – an Optimising ETC to IA32 Translator. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, number 59 in Concurrent Systems Engineering Series, pages 265–282. IOS Press, Amsterdam, The Netherlands, September 2001.

[7] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009.

[8] Peter H. Welch and David C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments – Proceedings of WoTUG 19*, pages 143–166, Nottingham-Trent University, UK, March 1996. World occam and Transputer User Group, IOS Press, Netherlands.

[9] 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, September 2004.

[10] Christian L. Jacobsen. *A Portable Runtime for Concurrency Research and Application*. PhD thesis, Computing Laboratory, University of Kent, April 2008.

[11] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[12] LLVM Project. (`http://llvm.org`).

[13] Rich Hickey. The Clojure Programming Language. In *DLS '08: Proceedings of the 2008 Symposium on Dynamic Languages*, pages 1–1, New York, NY, USA, 2008. ACM.

[14] Michel Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, Institut d'Informatique Fondamentale, 2005.

[15] David May. Communicating Process Architecture for Multicores. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 21–32, jul 2007.

[16] LLVM Language Reference Manual. (`http://www.llvm.org/docs/LangRef.html`).

[17] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

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

[19] Damian J. Dimmich and Christan L. Jacobsen. A Foreign Function Interface Generator for occam-pi. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, pages 235–248, Amsterdam, The Netherlands, September 2005. IOS Press.

[20] Paul Andrews, Adam T. Sampson, John Markus Bjørndalen, Susan Stepney, Jon Timmis, Douglas Warren, and Peter H. Welch. Investigating patterns for process-oriented modelling and simulation of space in complex systems. In S. Bullock, J. Noble, R. A. Watson, and M. A. Bedau, editors, *Proceedings of the Eleventh International Conference on Artificial Life*, Cambridge, MA, USA, August 2008. MIT Press.

[21] The Computer Language Benchmarks Game. (`http://shootout.alioth.debian.org/`).

[22] Kenneth R. Anderson and Duane Rettig. Performing lisp analysis of the fannkuch benchmark. *SIGPLAN Lisp Pointers*, VII(4):2–12, 1994.

[23] Tock Compiler. (`http://projects.cs.kent.ac.uk/projects/tock/trac/`).